深入剖析Linux内核与设备驱动

■ 董 峰 编著





本书配有源代码

下载网址 www.cmpbook.com



机械工业出版社
CHINA MACHINE PRESS

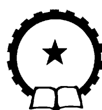


VISIT…

LANZAROTE Caliente.COM

深入剖析 Linux 内核与设备驱动

董 峰 编著



机 械 工 业 出 版 社

本书从需求的角度出发，以层次分析的方法探究 Linux 内核以及驱动的各部分框架和实现；以 TI 的嵌入式芯片为例，对内核各部分功能特别是电源管理功能进行代码级别的分析；以需求是什么、如何实现相应需求来进行整体分析，更易于对系统的把握和理解。本书将软件和硬件结合起来详细分析了嵌入式处理以及 Linux 内核实现中的各种技术。

本书可以作为本科和研究生操作系统的参考书，也适合 Linux 内核、驱动以及嵌入式系统各种级别的开发者和爱好者阅读。

图书在版编目（CIP）数据

深入剖析 Linux 内核与设备驱动/董峰编著 . —北京：机械工业出版社，2014. 9

ISBN 978 - 7 - 111 - 49426 - 3

Ⅰ. ①深… Ⅱ. ①董… Ⅲ. ①Linux 操作系统 - 程序设计 Ⅳ. ①TP316. 89

中国版本图书馆 CIP 数据核字（2015）第 035932 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
策划编辑：时 静 责任校对：张艳霞
责任编辑：刘 悦
责任印制：乔 宇
保定市中画美凯印刷有限公司印刷
2015 年 3月第 1 版 · 第 1 次
184mm × 260mm · 44. 25 印张 · 1098 千字
0001 - 2500 册
标准书号：ISBN 978 - 7 - 111 - 49426 - 3
定价：99.80 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：(010)88379833
读者购书热线：(010)88379649

网络服务
机 工 官 网：www. cmpbook. com
机 工 官 博：weibo. com/cmp1952
教育服务网：www. cmpedu. com
金 书 网：www. golden - book. com

封面无防伪标均为盗版

前　　言

笔者从研究生学习期间开始接触 Linux 内核至今已十年有余，直到现在仍然在研读各种内核书籍，细读之后始终有一些疑问和困惑。Linux 内核的需求从何而来，是怎样的需求？Linux 内核的层次结构是怎样的？每个功能模块是如何划分的？为什么要这么设计？多种设备及驱动应该如何划分，划分的依据是什么？设备模型究竟是怎么回事？电源管理技术是如何实现的？处理器与 Linux 内核之间的功能关系是怎样的？本书就是建立在对这些问题的思考和解答基础上的，读者可以在阅读本书的过程中找到这些问题的答案。这些问题的解答对于各种层次的开发者来说都是需要的，一方面，可以加深开发者对于系统的理解，做到明其理的程度；另一方面，从需求出发也符合对事物理解的规律，可加深对系统的认识。

笔者有幸于 2003 年加入 TI 开始嵌入式处理器开发之旅。TI 开放的文化使笔者有很多机会了解芯片的先进技术，TI 完备的开发文档使笔者可以了解各种实现细节，从而不断地成长。在进行了大量的代码注释工作后，笔者终于解答了之前的问题，从而写出了本书。

本书在结构编排上，从基础出发，使各章节相对独立，但是少量的向前或者向后引用还是必不可少的。总体上，本书是将最基本的章节尽量放到前面，所以推荐按顺序阅读。

在代码的引用上，以 TI 发布的 DM 3730 的 Android 版本中内核代码为主，为了突出主线部分和削减本书的篇幅，笔者以核心功能代码为主进行分析和介绍，而省略了辅助型代码。内核的代码是不断演进的，如果掌握了书中分析代码的思路，那么读者自己来对新版本的内核进行理解也不是不可能的。因为笔者水平所限，加之 Linux 内核本身就博大精深，所以书中肯定还会有一些错误，希望读者朋友们能不吝批评指正，以使大家可以共同提高。

读者如果在阅读本书的过程中有任何意见或者建议，欢迎通过下面的 E－mail 与笔者取得联系：dong1feng@ sina. com。关于本书使用到的源代码，读者可在 TI 网站上获取。

在本书写作过程中，父母和妻子给予了我很多生活上及精神上的支持，妻子还主动承担了校对的工作，谨以此书献给他们。另外特别提到岳父和岳母，是他们在女儿两岁前悉心的照料，使得我拥有独立而充足的时间进行学习和研究，在此表示由衷的感谢。感谢宝贝女儿花生，她的出生带来很多欢乐，也是我写作的动力之源。

还要感谢机械工业出版社的时静编辑，从选题的论证到文字编辑，他都付出了极其辛苦的劳动并且提出了很多有益的建议。

最后要感谢在 TI 遇到的兄弟姐妹们，用这本书来纪念我们一起战斗的日日夜夜。

董　峰

目　　录

前言
第 1 章　引言 …… 1
1.1　为什么要从硬件设备的角度看 Linux 内核 …… 1
1.2　从了解硬件开始 …… 2
1.2.1　最小系统 …… 2
1.2.2　完整设备介绍 …… 3
1.2.3　电源管理相关基础 …… 5
1.3　从设备看内核应该满足的需求 …… 6
1.4　所涉及的重要概念 …… 6
1.5　小结 …… 8
第 2 章　TI 应用处理器芯片及其内核特点 …… 9
2.1　DM 3730 微处理器 …… 9
2.1.1　DM 3730 微处理器框架 …… 10
2.1.2　DM 3730 微处理器特性 …… 11
2.1.3　DM 3730 微处理器电源管理相关设计 …… 12
2.2　DM 81XX 系列微处理器 …… 20
2.2.1　DM 81XX 系列微处理器框架 …… 20
2.2.2　DM 81XX 系列微处理器特性 …… 23
2.2.3　DM 81XX 系列微处理器电源管理相关设计 …… 24
2.3　Sitara 系列芯片 …… 28
2.3.1　Sitara 系列芯片框架 …… 29
2.3.2　Sitara 系列芯片特性 …… 31
2.3.3　Sitara 系列芯片电源管理相关设计 …… 31
2.4　TI 处理器内核特殊代码结构 …… 33
2.5　小结 …… 38
第 3 章　Linux 内核框架探究 …… 39
3.1　内核框架概述 …… 40
3.1.1　Linux 内核的层次分析 …… 40
3.1.2　Linux 内核模块间关联 …… 46
3.2　需求探究 …… 47
3.2.1　对内核核心的需求探究 …… 48
3.2.2　对设备管理的需求探究 …… 48
3.3　按需求的设备分类 …… 51
3.3.1　功能型设备 …… 55
3.3.2　总线型设备 …… 57
3.4　系统实现各种无关性的框架 …… 59
3.4.1　体系结构无关 …… 59
3.4.2　功能型设备的框架与总线无关 …… 62
3.4.3　总线控制器与总线设备的无关 …… 62
3.4.4　设备属性和设备操作无关 …… 64
3.4.5　策略和机制无关 …… 66
3.5　内核提供的基本服务和接口简介 …… 67
3.5.1　基本数据类型 …… 67
3.5.2　基本原子操作 …… 70
3.5.3　延时、调度、定时器相关 …… 71
3.5.4　锁操作 …… 72
3.5.5　抢占和屏障 …… 73
3.6　小结 …… 74
第 4 章　内核核心介绍及硬件的具体实现 …… 75
4.1　内核初始化 …… 75
4.1.1　内核初始化的基本需求 …… 75
4.1.2　内核初始化框架介绍 …… 76
4.1.3　TI 芯片内核初始化相关实现详解 …… 77
4.2　地址映射 …… 121
4.2.1　地址映射的基本需求 …… 121
4.2.2　地址映射框架介绍 …… 122
4.2.3　TI 芯片地址映射相关实现详解 …… 131
4.3　中断处理 …… 136
4.3.1　中断的基本需求 …… 137

4.3.2 中断处理框架介绍 …………… 138
4.3.3 TI 芯片中断处理相关实现详解 … 143
4.4 内存管理………………………… 161
4.4.1 内存管理的基本需求 ………… 161
4.4.2 内存管理框架介绍 …………… 162
4.4.3 TI 芯片内存管理相关实现详解 … 189
4.5 直接存储器访问单元（DMA） … 192
4.5.1 DMA 使用和管理基本需求 …… 192
4.5.2 DMA 使用和管理框架介绍 …… 193
4.5.3 TI 芯片 DMA 使用和管理相关实现详解………………………… 200
4.6 时钟（clock） ………………… 221
4.6.1 clock 管理基本需求 …………… 221
4.6.2 clock 管理框架介绍 …………… 221
4.6.3 TI 芯片 clock 管理相关实现详解 ………………………… 225
4.7 时间管理（Time） …………… 248
4.7.1 时间管理基本需求 …………… 248
4.7.2 时间管理框架介绍 …………… 249
4.7.3 TI 芯片时间管理相关实现详解 ………………………… 251
4.8 通用目的输入输出（GPIO） … 255
4.8.1 GPIO 管理基本需求 …………… 255
4.8.2 GPIO 管理框架介绍 …………… 256
4.8.3 TI 芯片 GPIO 管理相关实现详解 ………………………… 260
4.9 引脚复用（pin mux） ………… 275
4.9.1 引脚复用的基本需求 ………… 275
4.9.2 引脚复用框架介绍 …………… 275
4.9.3 TI 芯片引脚复用相关实现详解 ………………………… 276
4.10 小结 ……………………………… 290
第 5 章 内核设备管理以及驱动基础框架 ……………………………… 292
5.1 VFS 及其与设备的关联………… 292
5.1.1 VFS 框架 ……………………… 292
5.1.2 VFS 与设备关联 ……………… 303
5.2 Linux 设备模型（Linux device model） ………………………… 304
5.2.1 设备模型的需求及基本设计 …… 304
5.2.2 总线（bus） ………………… 311
5.2.3 驱动（driver） ……………… 314
5.2.4 设备（devices） …………… 317
5.2.5 功能类（class） …………… 324
5.2.6 设备资源管理（device resource） ……………………… 325
5.3 字符设备（char device） ……… 326
5.3.1 字符设备的特点和需求 ……… 326
5.3.2 字符设备的核心数据结构及操作 ………………………… 327
5.3.3 字符设备子类型 ……………… 333
5.4 块设备（block device） ……… 334
5.4.1 块设备特点和需求 …………… 334
5.4.2 块设备核心数据结构及操作 …… 335
5.4.3 块设备子类型 ………………… 346
5.5 电源管理………………………… 347
5.5.1 电源管理特点和需求 ………… 347
5.5.2 电源管理核心框架介绍 ……… 348
5.6 内核提供的同步操作、异步事件与单独执行实体的服务………… 366
5.6.1 同步操作服务 ………………… 366
5.6.2 异步事件……………………… 368
5.6.3 单独执行实体服务 …………… 375
5.7 内核提供的数据保护一致性操作服务……………………………… 380
5.7.1 数据保护一致性操作服务的需求 ………………………… 380
5.7.2 各种数据保护一致性操作简介 … 380
5.8 小结……………………………… 381
第 6 章 设备驱动之功能型驱动 ……… 382
6.1 输入设备（input） …………… 382
6.1.1 输入设备需求 ………………… 382
6.1.2 输入设备框架解析 …………… 382
6.1.3 输入设备应用层操作及框架适配 ………………………… 398
6.1.4 TI 芯片输入设备相关实现详解 … 402
6.1.5 输入设备电源管理相关说明 …… 407
6.2 帧缓冲（frame buffer） ……… 410

6.2.1 帧缓冲设备需求 …………………… 410
6.2.2 帧缓冲框架解析 …………………… 411
6.2.3 帧缓冲应用层操作及框架适配 …………………… 422
6.2.4 TI 芯片帧缓冲驱动相关实现详解…………………… 428
6.2.5 帧缓冲驱动电源管理相关说明 …………………… 443
6.3 音频设备（audio ALSA）………… 447
6.3.1 音频设备需求 …………………… 447
6.3.2 音频驱动框架解析 ……………… 448
6.3.3 音频驱动应用层操作及框架适配 …………………… 484
6.3.4 TI 芯片音频驱动相关实现详解 …………………… 493
6.3.5 音频驱动电源管理相关说明 …… 515
6.4 视频驱动（V4L2） ……………… 525
6.4.1 视频驱动需求 …………………… 525
6.4.2 视频驱动框架解析 ……………… 525
6.4.3 视频驱动应用层操作及框架适配 …………………… 554
6.4.4 TI 芯片视频驱动相关实现详解 …………………… 564
6.4.5 视频驱动电源管理相关说明 …… 581
6.5 小结…………………………………… 584
第 7 章 设备驱动之总线型驱动 ……… 585
7.1 内部集成电路总线（I^2C） …… 585
7.1.1 I^2C 总线驱动需求………………… 585
7.1.2 I^2C 总线驱动框架解析 ………… 586
7.1.3 TI 芯片 I^2C 总线驱动相关实现详解 …………………… 599
7.1.4 I^2C 总线驱动电源管理相关说明 …………………… 612
7.2 串行外设接口总线（SPI） …… 614
7.2.1 SPI 总线驱动需求………………… 614
7.2.2 SPI 总线驱动框架解析 ………… 615
7.2.3 TI 芯片 SPI 总线驱动相关实现详解 …………………… 625
7.2.4 SPI 总线驱动电源管理相关说明 …………………… 636
7.3 多媒体卡（MMC） …………… 637
7.3.1 MMC 需求 ……………………… 637
7.3.2 MMC 框架解析…………………… 639
7.3.3 TI 芯片 MMC 相关实现详解 …… 657
7.3.4 MMC 电源管理相关说明 ……… 667
7.4 通用串行总线（USB） ………… 669
7.4.1 USB 总线驱动需求 …………… 669
7.4.2 USB 总线驱动框架解析 ……… 671
7.4.3 TI 芯片 USB 总线驱动相关实现详解 …………………… 682
7.4.4 USB 总线驱动电源管理相关说明 …………………… 687
7.5 小结…………………………………… 689
第 8 章 设备驱动之 SoC 特殊驱动 …… 690
8.1 SoC 电源管理核心技术详解 …… 690
8.1.1 SoC 电源管理需求 …………… 690
8.1.2 TI 芯片 SoC 电源管理相关实现详解 …………………… 690
8.2 小结…………………………………… 699
参考文献 ……………………………… 700

第1章　引　　言

1.1　为什么要从硬件设备的角度看 Linux 内核

Linux 内核作为目前最成功以及发展最快的开源项目之一，在实际应用中取得的巨大成功是举世瞩目的。对庞大的 Linux 内核，大家是否有如下的疑问：究竟是哪些组织和个人为它的实现做出了贡献呢？又分别做出了多大的贡献呢？

Linux Foundation 于2013 年9 月发布了《Who Writes Linux》报告，公布了最新的不同组织对 Linux 内核贡献的情况，见表1-1。

表1-1　不同组织对 Linux 内核贡献

公　　司	修改次数	百　分　比	公　　司	修改次数	百　分　比
None	12,550	13.60%	NVidia	1,192	1.30%
Red Hat	9,483	10.20%	Freescale	1,127	1.20%
Intel	8,108	8.80%	Ingics Technology	1,075	1.20%
Texas Instruments	3,814	4.10%	Renesas Electronics	1,010	1.10%
Linaro	3,791	4.10%	Qualcomm	965	1.00%
SUSE	3,212	3.50%	Cisco	871	0.90%
Unknown	3,032	3.30%	The Linux Foundation	840	0.90%
IBM	2,858	3.10%	Academics	831	0.90%
Samsung	2,415	2.60%	AMD	820	0.90%
Google	2,255	2.40%	Inktank Storage	709	0.80%
Vision Engraving Systems	2,107	2.30%	NetApp	707	0.80%
Consultants	1,529	1.70%	LINBIT	705	0.80%
Wolfson Microelectronics	1,516	1.60%	Fujitsu	694	0.70%
Oracle	1,248	1.30%	Parallels	684	0.70%
Broadcom	1,205	1.30%	ARM	664	0.70%

贡献前30 名的公司中半导体厂商就有11 家，约占三分之一，如果算上硬件相关的公司会超过20 家，贡献总量超过了60%。可见硬件厂商对 Linux 内核的贡献是很大的，对 Linux 内核的影响也是巨大的。从这份表中笔者惊喜地发现，笔者之前就职的公司 Texas Instruments（简称 TI）也在其中，并且贡献量排名第四，在半导体厂商中仅在 Intel 巨头之后。在主要以 ARM 为核心的半导体的厂商中（如 TI、三星、高通、博通、英伟达、飞思卡尔），TI 的贡献则排名第一。

Linux 内核的开发需要硬件厂商的支持和参与，源于两个方面：一方面硬件是 Linux 内核工作的环境和基础，内核需要使用相应的硬件；另一方面 Linux 内核的发展趋势是在提高硬件的使用效率的同时保证设备功耗的最小化。这些都要求硬件厂商更多地参与到内核的开发以及设计工作中，只有更好地理解硬件才能更好地设计与实现内核。理解硬件能使我们更

好地理解内核的各个组成部分，在驱动方面理解硬件及其工作方式则更为重要。

1.2　从了解硬件开始

内核是执行在处理器之上的，从 Linux Foundation 发布的《Who Writes Linux》报告中我们已经看到，TI 在以 ARM 为核心的嵌入式处理器厂商中贡献排名第一，本书以 TI 的嵌入式处理器 DM 3730 为主介绍内核以及驱动的实现，同时会扩展到 TI 的 Davinci 系列和 Sitara 系列芯片。涉及到的知识可以扩展到所有处理器的内核以及驱动的实现，只是一些实现机制和细节不同罢了。

1.2.1　最小系统

我们先从系统的角度看看，最基本的硬件需求是什么？笔者认为是能执行，能进行运算。并不需要复杂的输入输出设备，只要能处理数据、能运行就可以。那么这种需求的最小系统是什么样子的呢？

图 1-1 是 LogicPD 公司基于 DM 3730 处理器的 SOM（System on Module）。可以看到非常的小，比硬币大不了多少，这个系统只要接上电池就可以运行了。不要以为这么小，能力就弱。Motorola 的里程碑系列手机使用的就是同样的处理器核心，是不是有些“007”设备的感觉呢！

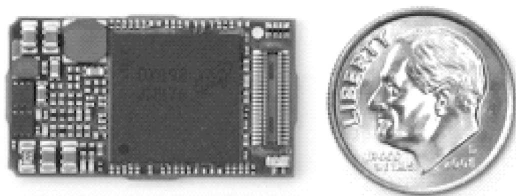


图 1-1　LogicPD DM 3730 SOM 示意图

再来看看 LogicPD 的 SOM 都有些什么，系统框图如图 1-2 所示。

主要的芯片就三个：

① PMIC：它负责供电。

② DM 3730：它是主处理器。

③ NAND Flash/mDDR SDRAM chip ：它是存储芯片，一个芯片中包括 mDDR SDRAM 和 NAND Flash。

大家注意到框图中的 PoP technology 了吗？PoP 是 Package on Package 的缩写，就是把两个芯片背靠背地焊在一起，这个非同一般，说明它们的关系非常亲密，要不为什么不把别的芯片 PoP 在一起呢。PoP 在一起的芯片分别是主处理器 DM 3730 和 NAND Flash/mDDR SDRAM chip。主处理器和内存是所有系统必需的，关系自然紧密了，这是能 PoP 的资本。PoP 的优势很明显就是使电路板的面积减小了，这个对于手机等对电路板大小要求高的设备来说是非常重要的。

另外需要注意两点：其一是框图的左上从 Connectors 引入的 power，其二是两个时钟

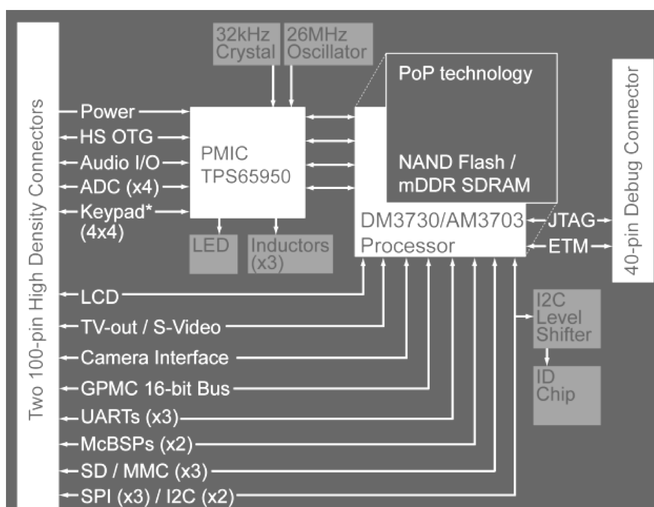


图 1-2　LogicPD DM 3730 SOM 系统框图

（一个是 32 kHz Crystal，另一个是 26 MHz Oscillator）。为什么要用两个时钟，后续讲解电源管理实现时会讲到。注意这两点主要是想向大家说明，数字系统中，power 和 clock 的概念一定要有，任何数字芯片以及芯片内部的数字模块都需要 power 和 clock，没有了两者中任何一个，相应的数字部分都无法工作。换句话说，要想使数字模块工作，首先要提供 power 和 clock。power 和 clock 也会贯穿整个电源管理以及内核的驱动中。这是本书强调的第一个概念，也是非常重要的概念。

剩下的就是和 Connectors 连接的输入输出设备，后续章节讲驱动的时候都会涉及。

笔者和朋友一起做的 SOM 也是这种类型。只是将 NAND Flash/mDDR SDRAM chip 换成了 EMMC/mDDR SDRAM chip，虽然不能通过 PoP 技术来减小电路板面积，但是可以使用大容量的 EMMC 来支持 Android 4.3（NAND 对 Android 来说容量太小），从这个角度来说算是一件好事。

1.2.2　完整设备介绍

只是 SOM 的话没有太多使用价值，真正的设备是要连接很多复杂外设的，以 DM 3730 为基础的设备框图如图 1-3 所示。

图 1-3 引自 TI 的《DM 3730 芯片手册》中第 136 页框图，为了方便读者在芯片手册中查找相关的内容，会在引用芯片手册框图时说明其所在的页码。笔者从对硬件毫无了解的计算机软件专业毕业生到目前个人感觉对嵌入式真正了解的开发者的成长过程中，TI 的芯片手册给了我很大的帮助，其中详细讲解了诸多的原理和实现细节。当然很多人会觉得几千页的芯片手册无从下手，但是对钻研技术的人来说这些可是宝贝。如果大家可以静下心来仔细品味，绝对会受益匪浅。

图中，DM 3730 通过各种连接方式连接了各种设备，输入输出设备根据不同的类型大体可以分为电源管理、用户输入、显示输出、图像采集、存储以及无线设备等。我们可以将 DM 3730 与这些设备的数据接口分为总线和单一的数据接口总线。总线（如 I^2C、SPI 和 USB）的显著特点是单个总线上可以连接多个设备（如 User Interface 部分 Finger Print 和 Touch Screen 都是通过 SPI 总线和 DM 3730 进行连接）；单一的数据接口只连接单一类型的设备（如用于用

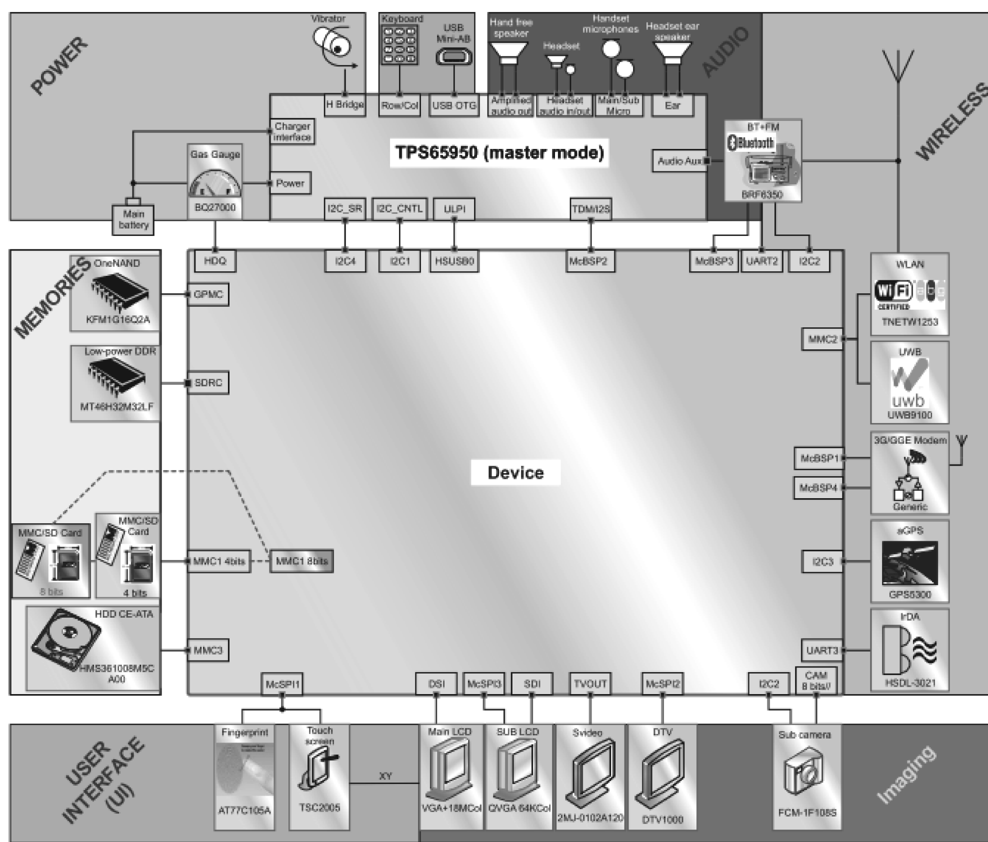


图 1-3　基于 DM 3730 的设备框图

户显示 LCD 输出的 DSI 接口）。各种不同类型的数据连接接口其设计思想以及功能是不同的，例如 I^2C 通常用于设备的控制，McBSP 用于音频数据的传输等。需要注意的是同一总线可以连接不同类型的设备，比如 MMC 可以连接 SD 卡也可以连接 WIFI 设备。这些接口都是长期发展的行业标准，是需要软件适应并满足的需求。DM 3730 作为 SoC，其中包含图中所有接口控制器，相应的控制器用于按照相关的总线和接口标准进行数据传输。驱动的开发人员必须了解对应的总线以及接口知识，相关知识理解越深刻，解决问题就越游刃有余。

为什么要有这么多连接方式呢？这和数据传输的需求、数据的特性及复杂程度相关，比如数字信号通常依靠 clock 信号同步，那么相应的带宽就是 Fclock × bits，bits 为并行传输的位数，总线设计的时钟频率范围不同、位数不同相应的带宽就不同，而频率和位数不能随意提高，高速信号会产生电磁效应影响其他信号的完整性，另外从需求的角度来讲，不是所有类型的数据都需要高速传输，比如控制数据可以通过 I^2C 传输，这种 2 线低时钟扩展性较好的总线，能够方便的进行硬件设计并通过其连接各种各样的传感器，丰富我们的实际生活。在高速数据总线设计方面，信号频率越来越高，并行信号在高主频时会有先天的劣势，很难保证信号完整性，而差分信号则可以避免相应的问题，现如今视频输入输出、高速硬盘、PCI 甚至连外部 memory 都逐渐转向差分信号的传输方式，伴随而来的问题就是对相关接口的调试会复杂一些，需要理解协议。另外信号的分析需要专门的设备，这和调试并行信号只需要示波器比起来就显得复杂得多了。总之对接口来讲，最需要了解数据是如何组织传输的。

1.2.3 电源管理相关基础

目前嵌入式设备很多都是电池供电，这类设备电池续航能力成为人们重点关注的指标之一，这就引出了一个新的技术方向——电源管理技术。

首先了解一下嵌入式芯片功耗和哪些因素相关：

The active power for a CMOS device is defined as：$P = CV^2F$，where P = active power needed for switching，C = total capacitance being switched，V = operating voltage and F = switching frequency。

这段英文说明了影响功耗的因素，C 主要和芯片的逻辑单元的状态相关，通常逻辑单元关掉时 C 的值比较低；V 是操作电压；F 是工作频率。注意功耗的需求是在稳定工作的前提之下的，而从公式中明显看出影响最大的因素是 V。围绕着 C、V 和 F 会有很多电源管理技术的实现，后续会有详细的解析。

接下来以 DM 3730 为例了解一下主芯片和电源管理芯片的连接，如图 1-4 所示。

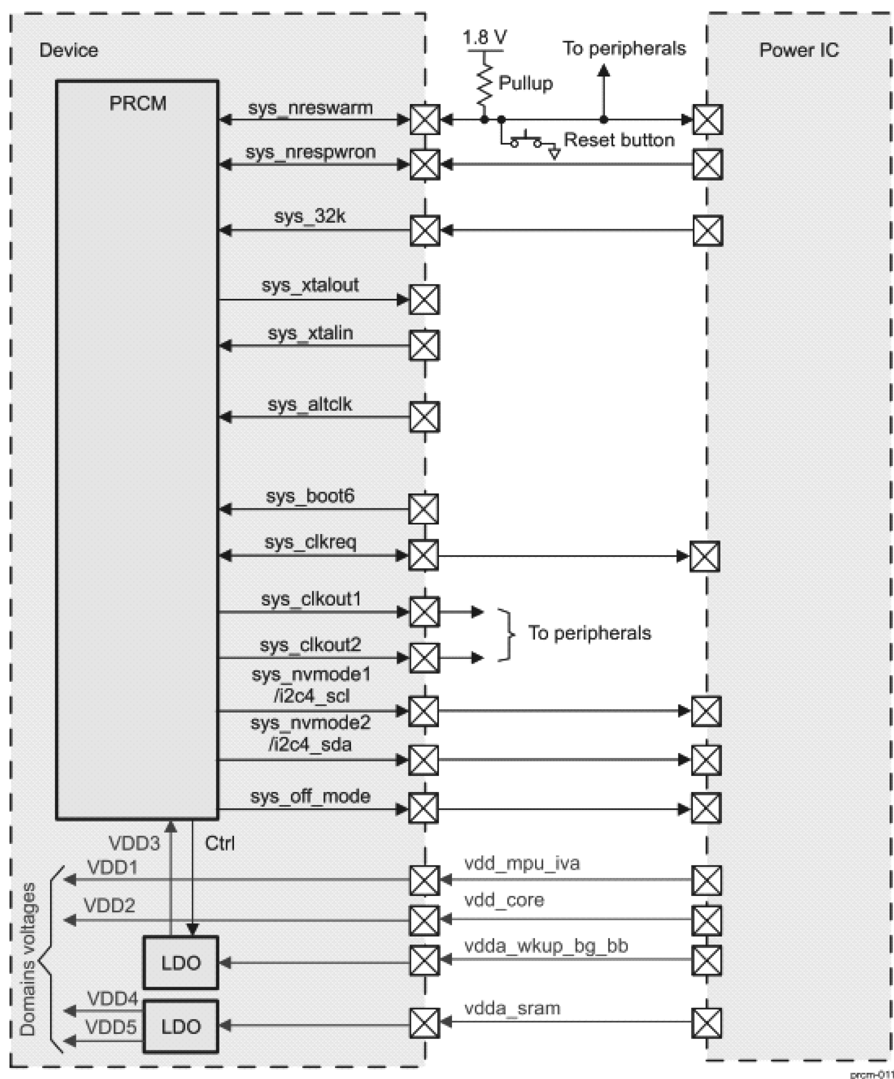


图 1-4 DM 3730 与电源管理芯片连接框图

图 1-4 是引自《DM 3730 芯片手册》中第 239 页的框图，其中 PRCM 是 power reset clock management 的缩写，它是对电源、启动信号和时钟进行管理的模块。从图中我们看到时钟信号（如 sys_32k）和多个电源（如 vdd 开头的连接）都是由电源芯片（Power IC）提供的，为什么要提供多个电源呢？这是因为 SoC 中不同的工作单元需要的电压不同，如果都要求高电压，根据之前的公式，功耗影响会比较大，通过隔离不同的工作电压可以降低整体的功耗。电源管理技术的大部分操作就是围绕着这几路电源和时钟展开的。

这里我们看到一路电源的名字是 vdda_sram，这说明芯片内部有 sram，这部分 memory 在系统启动和电源管理的一些特殊操作时起到至关重要的作用，另外这部分 memory 访问效率非常高，可以用来提高系统性能。

在图中还有诸多疑问，比如 PRCM 详细内容，具体功能是什么？两个 I^2C 接口的作用是什么？多路电源（如 vdd_mpu_iva 等）具体功能是什么？其他部分的信号起什么作用？这里先了解基本概念，这些疑问会在后续电源管理的详解中进行说明。

1.3 从设备看内核应该满足的需求

对硬件有了基本的了解，就可以从硬件出发考虑内核应该满足的需求了。

硬件的类型是多种多样的，连接的接口也是变化多样的，作为内核的一个最基本的要求就是能够访问并操作这些设备，这就需要大量的设备驱动支持。同时也要支持各种不同的数据连接接口。内核也需要支持各种总线，并且要支持各种类型的设备，由于总线要符合一定的协议并连接多个设备，所以通常分为总线控制的主设备以及连接进行总线响应的从设备；内核会应用到各种设备，所以需要同时为各种总线的主设备以及从设备提供支持。当然电源管理的需求会涉及以上各种设备、总线和接口。另外内核不能只是支持特定的处理器，需要支持各种类型的处理器，学术一点的说法就是要支持各种体系结构。

还有一点比较重要的思路就是，大型的系统会涉及很多人共同开发，这就对代码的可维护性和重用性提出了很高的要求，针对这个需要在设计过程中内核会将各种共同的资源抽象出来统一管理，并考虑硬件无关性形成相应的模块或者函数接口以供开发者使用。

总结下来从设备出发看内核要满足以下需求：

- 能支持不同的体系结构和处理器。
- 能支持不同的总线连接以及总线设备。
- 能支持不同的数据连接接口以及相应的设备。
- 能支持各个级别的电源管理功能。
- 设计时要考虑硬件无关性提高各模块的重用性。

前四项都是比较直接的需求，第五项则是考验开发人员的设计能力，也是内核的各个模块的设计重点以及我们研究的重点。

1.4 所涉及的重要概念

谈到 Linux 内核，大家的第一感觉就是一个庞大的系统、很多的模块和功能，实在是太复杂了。究竟如何入手是个问题。对系统的理解有很多的方法：自顶向下、自底向上等。笔

者更认同自顶向下按层次分析的方法。顶部是用户的直接接口和需求体现，按照需求从整体到局部会更容易理解系统，也更容易设计与实现系统。整本书的论述也是从需求出发试图按照 What（需求是什么）——→How（如何实现）的逻辑解释系统、子系统及各个模块。

本书中还会涉及一些重要的概念，如资源、数据、数据流、接口、临界区、锁、执行实体、上下文、同步、异步、时间、空间和抽象等。需要读者对这些概念有基本的认识。下面重点介绍这些概念与需求之间是怎样的关系。

首先需求分为功能需求和性能需求，这两种需求都会影响系统的实现。

在实现需求的时候要考虑资源，有哪些资源可以使用，如何使用来满足需求，这些都是要考虑的。需要注意的一点是资源通常都是共享的，对某个共享资源的管理使用通常就会抽象成系统的一个模块。对硬件来说，处理器、memory、clock、power supplier、DMA、中断等都是资源。

在系统中各种资源围绕什么进行操作呢？是数据。数据可以加上不同的限制条件，换句话说数据可以组织成不同的形式。数据经由不同的模块处理可以理解为数据在不同的模块之间流动，笔者称之为数据流。数据流经过相邻两个模块时，双方要知道数据是如何组织的，从而才能形成接口。系统的设计与实现中一定要有数据流的概念还要明确接口的数据组织形式。这样才更容易理解系统是如何运作的，以设计出更好的系统。另外当不同的实体共享数据时，需要对共享数据进行保护，避免不同实体同时访问共享数据，这个保护区域就是临界区，保护方法是加锁。

对处理器这一资源的使用要有执行实体的概念，执行实体的运行受处理器的执行级别和运行状态的影响，软件通常是通过不同的栈来区分级别（如进程栈和中断栈），Linux 内核中断实现是借用当前进程的内核栈来执行。不同的执行实体与处理器的执行级别和状态一起形成上下文的概念。不同的上下文在临界区时需要不同级别的锁。

其次对需求的实现方法通常有同步、异步两种方式，需要注意的是同步、异步概念在软件和硬件上是不同的。软件中同步是指发送方发出数据后，等接收方发回响应以后才发下一个数据包的通信方式；异步是指发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通信方式。硬件中同步信号和时钟信号有关，实际上输入信号和时钟信号进行了与运算或与非运算，输入信号和时钟信号的运算结果为有效状态时，器件的状态才会改变；异步输入信号和时钟信号无关，输入信号变为有效状态时，器件的状态就会立即改变。可见软件和硬件对同步、异步的理解所站的立场不同，软件站在发送者的立场，硬件则是站在接收者如何确认数据有效的立场，这与软件和硬件的不同特点是分不开的。这些概念我们要有，并且需要在驱动的设计与实现中随时站在不同的立场上考虑问题。需要注意：软件、硬件的同步、异步的概念虽然有区别，但是也有关联。比如软件中，异步概念的实现很多时候是通过硬件的异步事件（如中断）来实现的。

在实现需求时还要考虑时间、空间的概念。这里时间主要是性能需求，空间主要是对存储资源的占用。计算机专业的学生经常会听到老师讲“时间换空间，空间换时间”，这句话体现了效率和资源之间置换的思想，如果想提高效率通常就要多占用资源，相应的如果资源是瓶颈就要牺牲效率。所以在系统设计实现中一定要首先搞清楚效率和资源的关系，如果本末倒置，系统必定失败。Linux 内核在设计过程中这些都已考虑在其中，这也就提升了它的适用范围。

最后，软件实现需要进行抽象。抽象是人类的一个重要思维能力，从某种角度来说软件系统是对各种概念或者行为进行抽象，并加以管理。抽象在系统设计中主要体现在将共性提取形成概念，抽象的概念通常包括属性及其相应的行为，这在内核的设计中处处可见，比如 file 结构为各种文件的抽象。各种硬件资源在内核中也会有相应的抽象实体存在。我们在学习内核的过程中经常考虑这些抽象概念包括哪些属性以及如何操作，就像走了一遍内核的设计过程。比起一味的死记硬背会更好地帮助我们理解系统。

1.5 小结

本章从观察硬件设备出发来探讨内核应该满足哪些需求，让大家对硬件以及电源管理有一个基本的认识，大致理解内核应该满足的需求。最后介绍了本书涉及的重要概念与需求的关系，以帮助大家对后续内容进行学习。

第 2 章　TI 应用处理器芯片及其内核特点

在后 PC 时代，嵌入式系统得到了空前的发展，嵌入式产品已经随处可见。随着网络技术和移动通信的发展，手机作为高端嵌入式系统更是成为人们日常生活中必不可少的工具。嵌入式处理器是嵌入式系统的核心，单一的嵌入式处理器是无法满足小到电子玩具大到飞机控制系统的所有需求的。面对各种各样的需求，嵌入式产品厂商的首要工作就是选择合适的芯片实现自己的系统，换句话说就是嵌入式产品的设计与实现要从“芯”开始。了解嵌入式系统也要从“芯”开始。

随着时间的推移和技术的进步，在工业控制和新兴的手持式应用等领域，用户体验成为产品成功的关键因素之一，越来越多的产品需要良好的用户界面、互联功能以及较强的数据处理能力。TI 公司作为老牌的嵌入式处理器提供商，为了应对嵌入式中、高端应用领域，如工业控制、POS 机、网络设备、图像处理、手机等不同的需求，于 2008 年之后分别推出了 DM 3730 系列、DM 81XX 系列和 Sitara 系列芯片。这些芯片各有特色，它们可以涵盖嵌入式中、高端应用的各个领域。

2.1　DM 3730 微处理器

提到 DM 3730 芯片就不得不提到 OMAP3 系列芯片，光说 OMAP3 知道的人会比较少，但是提到 Motorola 的 Milestone 手机知道的人就多了。Milestone 手机正如其名开创了 Android 智能手机的里程碑。OMAP3 系列芯片在手持设备的应用见表 2-1。

表 2-1　OMAP3 系列芯片在手持设备的应用

名　　称	制造工艺	处理器指令集	处理器类型和主频	应 用 终 端
OMAP 3430	65 nm	ARMv7	600 MHz ARM Cortex - A8	Motorola Droid/Milestone, Nokia N 900, Palm Pre, Samsung i 8910, Sony Ericsson Satio
OMAP 3440	65 nm	ARMv7	800 MHz ARM Cortex - A8	Archos 5 (Gen 7), Motorola Milestone XT 720, Motorola Titanium XT 800, Samsung Galaxy A (SHW - M 100S), Samsung i 7680
OMAP 3630	45 nm	ARMv7	600 MHz ~ 1.2 GHz ARM Cortex - A8	3630 - 1000: Archos 43, Archos 70, Archos 101, LG Optimus Black, LG Optimus Bright, LG Optimus Mach, Motorola Cliq 2, Motorola Droid 2 R2D2 Special Edition, Motorola Droid X, Motorola Defy +, Nokia N 9, Nokia N 950, Palm Pre 2, Panasonic P - 07C, Panasonic Sweety 003P, Samsung Galaxy SL I 9003, Sony Ericsson Vivaz, Lenovo A1-07, Samsung Galaxy Player 4.2 (YP - GI1), Le Pan TC 970

从该表中可以见证 OMAP3 系列芯片的成功，各大手机厂商都应用该系列芯片推出了手机产品，这也说明 OMAP3 系列芯片非常适用于手持设备，是手持设备芯片的典范。

DM 3730 实际上使用的是 OMAP 3630 的 DIE，封装上进行修改可以让生产厂商使用 0.8 mm 的技术进行焊接，这样的改进使得更多的厂商可以开发和生产高性能、低功耗的手持设备。

2.1.1 DM 3730 微处理器框架

DM 3730 微处理器框架如图 2-1 所示。

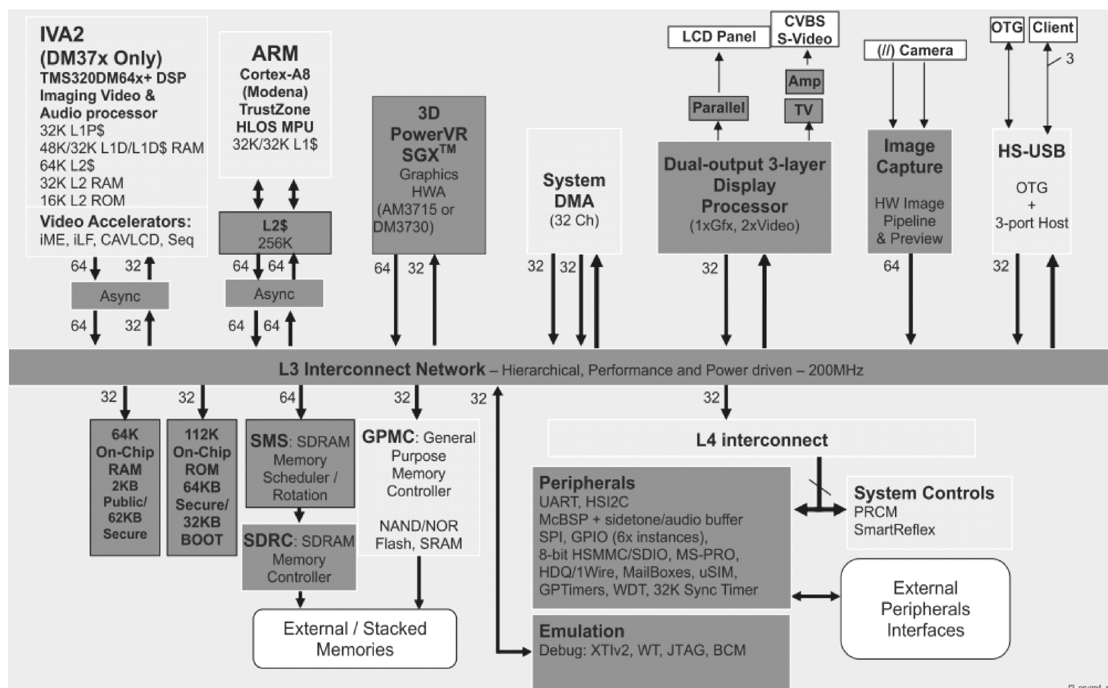


图 2-1　DM 3730 系统框图

图 2-1 引自 TI 的《DM 3730 芯片手册》中第 187 页框图。从框图中可见 DM 3730 的主要核心与外围接口。

DM 3730 有三个核心处理器单元：

① ARM Cortex - A8 ：负责系统控制和外围接口驱动。Linux 运行在该处理器上。

② IVA2 ：负责音视频编码解码或者信号处理的加速单元。其中包含 C64x + 核心 DSP 以及视频加速单元，具有 720P@30fps 的编码或解码能力。

③ PowerVR SGX：负责 3D 渲染的硬件加速单元。

主要外围设备的接口如下：

① Dual - output 3-layer Display Processor：负责处理 UI 的显示可以实现三层画中画的硬件叠加，并支持同时两个设备进行显示。

② Image Capture：负责图像采集可以进行图像信号处理。

③ HS - USB：负责 USB 的连接，可以通过 USB 与 PC 连接或者与 USB 设备连接。注意其中有 3-port Host，最多可以连接三个 USB 设备，很多手机厂商都是通过两个 USB Host 接口实现双网双待手机的。

④ SDRC：负责 DDR memory 内存的连接。

⑤ GPMC：负责各种 Flash 存储器的连接，该接口可以通过时序编程与 FPGA 进行连接以扩展功能。

⑥ Peripherals：各种外设接口，如 UART（串口）、I^2C（通常负责传感器的控制）、McBSP（负责音频数据传输）、HSMMC/SDIO（负责 SD 卡、eMMC 或者 WIFI 芯片的连接），uSIM（负责 SIM 卡的连接）等。

⑦ GP Timer：可编程的定时器，向系统提供定时功能。

除去以上的模块之外还有模块：

① System DMA：负责数据在外设和内存或者内存不同区块之间的复制，用来降低处理器的负载以提高系统性能。

② On－Chip RAM：片内 RAM，使系统在外部内存没有初始化好的情况下仍可运行。通常该段内存可用于系统初始化，系统加速以及电源管理相关的功能。

③ On－Chip ROM：类似于 PC 上的 BIOS，其中包含系统的初始化程序。

④ L3 Interconnect Network：片内高速总线，负责在片内各个模块间建立高速数据通道。

⑤ L4 Interconnect：片内低速总线，负责隔离高速设备与低速设备。

从 DM 3730 的框架来看，它非常适合手持设备的应用，手持设备通常不会连接硬盘这种高功耗的大容量存储设备而是用 SD 卡进行替代，DM 3730 中就设计了三个 HSMMC 接口而去掉 SATA 接口以适应这种需求。在总线方面 PCIE 由于功耗高、插槽大不适用于手持设备的应用，USB 则是优选，相应的 DM 3730 中设计了一个 USB OTG 接口和三个 USB Host 接口满足这种需求。另外为了降低功耗，DM 3730 的外部内存接口设计成 LPDDR 接口，I/O 工作电压也设计成 1.8 V，可见为了满足高性能低功耗的需求，DM 3730 在框架设计方面已经做足了工作，这也是它能流行的一个原因。

2.1.2 DM 3730 微处理器特性

DM 3730 支持高性能手持设备的典型应用，其特性如下：

- DM 3730 核心处理器经过配置可在支持低功耗或高性能模式的多个工作点下工作。
- 高达 800 MHz 的 TMS320C64x＋DSP 支持 720P@30fps 的高清视频编码和解码；DSP 引擎可编程，支持多个一般性信号处理任务，如数字滤波、数学函数以及影像处理与分析等。
- 具备 PowerVR 200 MHz 图形加速器，支持 OpenGL ES 2.0，每秒可渲染 2 千万个多边形，并具有高级显示子系统。
- 1.8 V 的 I/O 接口降低 I/O 功耗适合电池供电的系统。
- 芯片待机功耗低于 0.1 mW。

为了支持手持设备各种情况下功耗的需求，DM 3730 电源管理方面的特性如下：

- 内部时钟管理。
- 内部启动信号管理。
- 唤醒事件管理。
- 智能反馈电源管理。
- 动态电压/频率管理。

● 待机功耗管理。

2.1.3 DM 3730 微处理器电源管理相关设计

1. DM 3730 使用的电源管理技术介绍

电源管理的相关技术都是以第 1 章讨论过的公式 $P = CV^2F$ 为原理出发的。DM 3730 中使用了两个比较重要的电源管理技术 DVFS（动态电压/频率调整）和 AVS（电压适配调整）。先看看 DVFS，从公式出发可以知道降低频率就可以降低功耗，然而直接降低频率对相同工作量的任务就需要更多的时间完成，如此看来并没有起到降低功耗的作用，执行时间变长了，用户体验也不好。从公式出发看看降低电压的效果，由于降低电压对芯片工作稳定性会有影响，所以一般在降低电压的同时是要降低频率的，这样一来完成相同的任务也会消耗更多的时间，所以降低电压会是一个连锁反应，但是从公式可以看到电压对功耗的影响是 V^2，可见电压影响更大。举个例子电压为 1 V 时执行 1 s 的任务如果消耗 1 J 的能量，那么主频降为原来的一半时所需要的电压大概是 0.7 V，这样同样的任务大概需要 2 s，这样算下来消耗的功耗大概是 0.5 J，总体算下来降低电压和频率能使功耗大大降低，但是这种功耗的降低的代价就是要用时间来换，DVFS 就是在不同的 OPP（operating point，操作点）之间调节，每个 OPP 对应一组电压和频率。既然 DVFS 是通过时间来换取功耗的降低，那么有没有无代价的降低功耗的技术呢？答案是有的，这就是 AVS，AVS 从字面adaptive voltage scaling 理解是电压适配调整，究竟怎么电压适配呢？图 2-2 说明了 AVS 的基本原理，适配这个词突出了芯片的个性和环境的差异，适配实际上就是适配芯片个体情况，对不同芯片的工作电压一定要保证适应最差的情况，如图 2-2 中 OPP1、OPP2 和 OPP3 连成的线，只有使用最差情况的设置才能满足所有芯片的需求。情况好的芯片，如果也要在最差情况的电压下工作，这就会产生功耗的浪费，所以需要适配来降低功能。

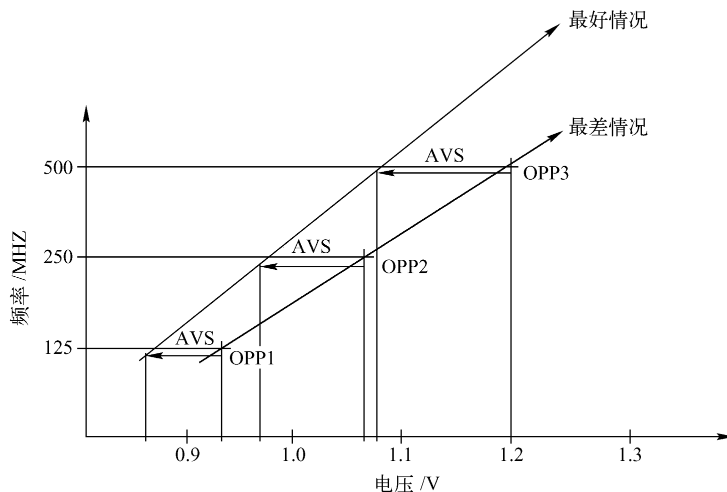


图 2-2　电压适配调整原理

针对 AVS 的需要，TI 设计了 SmartReflex 智能反馈电压控制技术，实现框架如图 2-3 所示。该图引自 TI 的《DM 3730 芯片手册》中第 234 页框图。

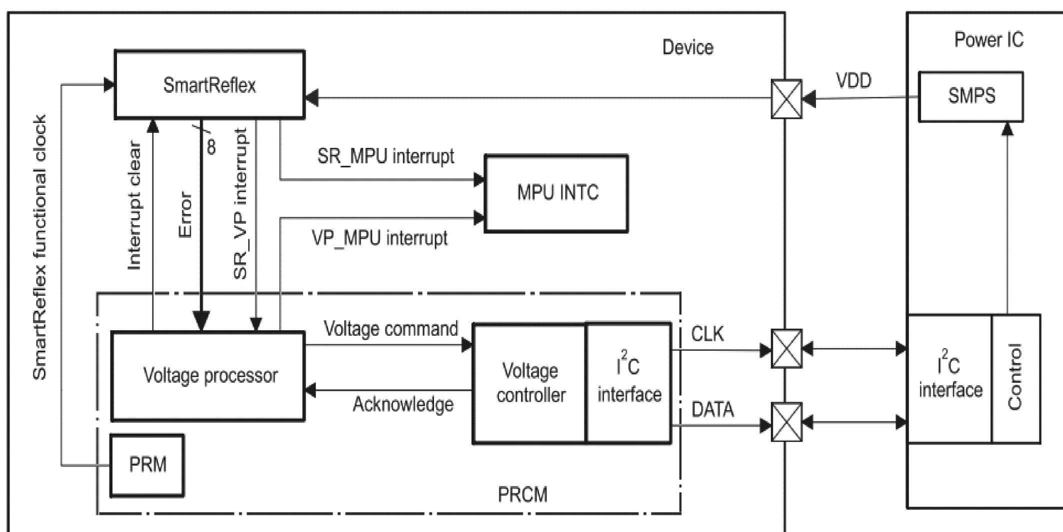


图 2-3　SmartReflex 实现框架

从图 2-3 中可见，有 Voltage processor（电压控制器）对当前芯片的个性数据进行计算，然后通过特定的 I^2C 接口和电源管理芯片通信，控制电源管理芯片的电压输出，进而为芯片提供合适的电压。SmartReflex 会通过对输入电压进行检测以及提取芯片自身工作状态的数据为电压控制器提供操作的依据，这样就形成一套完整的反馈系统，合理根据芯片自身的工作情况调整工作电压使功耗最低，从而降低芯片的功耗。

电源管理技术中，DVFS 是对电压的大范围的调整，而 AVS 是针对某个 OPP 情况下电压小范围的调整，两者完美的结合就能实现功耗最大的优化。

2. DM 3730 电源管理模块详解

（1）与电源管理芯片相关的设计

DM 3730 中实现电源管理技术的模块是 PRCM，PRCM 是 power reset clock management 的缩写，从名字就可以看出该模块管理芯片内部的电源、启动和时钟信号。PRCM 的系统框图如图 2-4 所示。

图 2-4 引自 TI 的《DM 3730 芯片手册》中第 237 页框图，在该框图中我们明显看到有 5 个 VDD，每个 VDD 对应一个 voltage domain（电压域），它提供一种电压规格，可以为多个有相同电压需求的模块提供电源，voltage domain 也可以理解为完整的物理供电线路。设计多个 VDD 的好处是可以隔离不同的电源需求，在 DM 3730 特性中已经介绍，它的待机功耗可以做到低于 0.1 mW，要实现这一点就必须将待机时工作的模块最小化，并且其他模块不能有漏电流，由于只要有电压差就会有漏电流，这就需要主要的模块在待机时电源完全关掉使电压降为 0 V。为了达到如此低的功耗，图 2-4 中的 VDD1 和 VDD2 在待机时都要降为 0 V。根据第 1 章的图 1-4，可知 VDD1 ~ VDD5 都是来源于电源管理芯片，所以如果要改变任意一个 VDD 的电压都是需要和电源管理芯片配合，由电源管理芯片完成实际的操作，可以说电源管理芯片就是按照 DM 3730 的要求进行操作，同样的 VDD1 和 VDD2 降为 0V 也是 DM 3730 请求电源管理芯片完成的，这个请求在 DM 3730 设计中采用了最简单的办法即通过一个引脚（图 2-4 中的 sys_off_mode）极性的改变就完成了。在第 1 章有个遗留问题是关

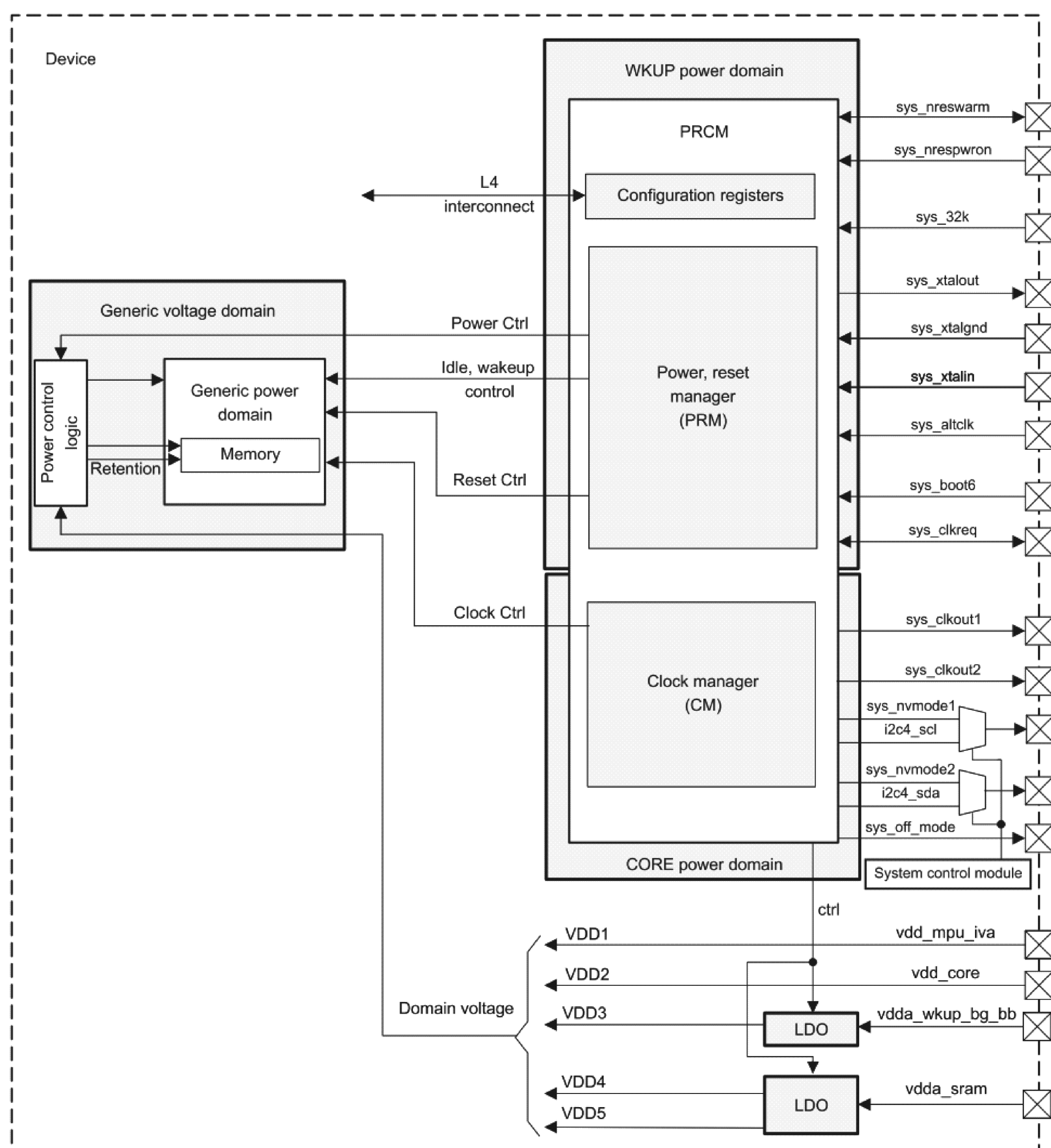


图 2-4　PRCM 系统框图

于最小系统框图中为什么有两个时钟（一个是 32 kHz Crystal，另一个是 26 MHz Oscillator）。现在可以说明一下，图 2-4 中实际也有两个时钟信号，分别是 sys_32k（接入 32k 时钟信号）和 sys_xtalin（接入 26M 时钟信号），在提供系统内部时钟的时候，是需要锁相环进行倍频的，由于内部最高达到 1 GHz 的频率，所以需要 26 MHz 时钟来倍频，可以说 26 MHz 时钟是内部所有时钟的“父亲”。这样就遇到一个问题，在待机的时候如果还是使用 26 MHz 这种高速的时钟，那么功耗就会有比较多的浪费，DM 3730 设计的时候考虑到该问题，当待机时只要提供 32 kHz 的时钟就能保证必要的模块待机，从而尽量减少功耗。由于时钟也是电源管理芯片输出的，如果要停掉高频时钟也需要一个信号，这个信号就是图 2-4 中的 sys_clkreq 信号。可以看到 DM 3730 的很多电源管理功能还是需要电源管理芯片的特殊支持，所

以要达到好的电源管理效果需要能支持这些特殊功能的电源管理芯片。TI 为 DM 3730 设计了一系列电源管理芯片如 TPS 65950 等，这样使用 DM 3730 再加上 TPS 65950 的配合就能够达到很好的待机功耗，TI 的参考设计中也是这么实现的。

（2）芯片内部电源控制设计

图 2-4 中的左侧可以看到两个名词：voltage domain 和 power domain，voltage domain 中包含 power domain。voltage domain 已经做过介绍就是电压域，power domain 从名字可知是和 power 相关的概念，它是电源控制的基本单元，控制整个域电源的开关。在优秀的电源管理设计中，不使用的模块应该是可以完全切断电源的，这样可以达到降低功耗的目的。power domain 正是基于这个概念设计的。由于设计的复杂度也是要考虑的因素，权衡下来，把那些能够一起开关的模块放在一起，形成一个 power domain。可见，power domain 实际上是个逻辑的概念，逻辑上一起开关的模块都可以做到 power domain 中，这样有一个问题就是 voltage domain 是和物理电压相关的概念，而 power domain 则是逻辑的概念，那么这两个概念之间的关系就不是纯粹的包含与被包含，而是相互重叠的概念，比如一个 power domain 可以在不同的 voltage domain 中，这些都取决于设计需求。有了 power domain 之后，系统在不需要某些功能的时候就可以将相应的 power domain 完全关断以节省电能，可以说是按需取能，该功能不仅在电池供电的设备中需要，在提倡节能环保的今天可以说所有的电子设备都需要，为了能使设备的某些功能按照需要进行开关，Linux 设计了 runtime power management 框架。

回过头来继续讨论 voltage domain 和 power domain 在 DM 3730 中的实现，图 2-5 引自 TI 的《DM 3730 芯片手册》中第 371 页框图，其中描述了 DM 3730 中 voltage domain 和 power domain 的具体关系。

图 2-5 中需要注意两个部分：

其一，就是 VDD3 voltage domain 和其中的 WKUP domain。从图 2-4 中可以看出 VDD3 是由 vdd3_wkup_bg_bb 单独提供的一路电源，在图 2-5 中可以看出只有 VDD3 voltage domain 中包含唯一的 power domain 即 WKUP domain，也就是说 WKUP domain 单独由一路电源供电，可见 WKUP domain 的重要性。WKUP domain 之所以重要是因为在整个 DM 3730 的各个 power domain 中需要一个一直保持上电的模块，这个模块需要功能最小，在需要低功耗的时候其他 power domain 都关闭，留下该 domain 当守卫，监视外部环境。如果有外部信号需要系统启动工作，该模块还需要唤醒系统，WKUP 是 wake up 的缩写，表示该模块上电之后一直保持清醒，可见这是一个很辛苦很重要的模块。WKUP domain 中包括一组 GPIO、一个通用定时器和一个看门狗定时器，这些都是在系统进入待机状态后能唤醒系统所需要的基本硬件，从中可以看出要设计有良好的电源管理功能的芯片需要考虑很多细节。

其二，就是框图中有些部分在两个 voltage domain 内，例如很多模块的 SRAM 分为 Logic 和 Array 两部分，并且分别在两个不同的 voltage domain 内。SRAM 是每个模块内部的 memory，比如 cache、line buffer 等。这部分的逻辑和存储分开是有好处的，可以保证存储的内容和访问操作完全分离，保证逻辑和存储可以单独设置电源管理等级，可以保证在信息不丢失的情况下尽量减少功耗。另外对逻辑（Logic）部分会根据不同的 OPP 调节电压，而对 memory 部分则没有这种需求，这也要求逻辑和存储部分的电源分离。DPLL 部分也在两个 voltage domain 中，则是由于最终的输出时钟的使用者是有不同的 OPP，如果整个 DPLL 的逻辑完全支持不同的 OPP，设计复杂度要高，另外 DPLL 的时钟输入没有多个电压规格的要求，这样

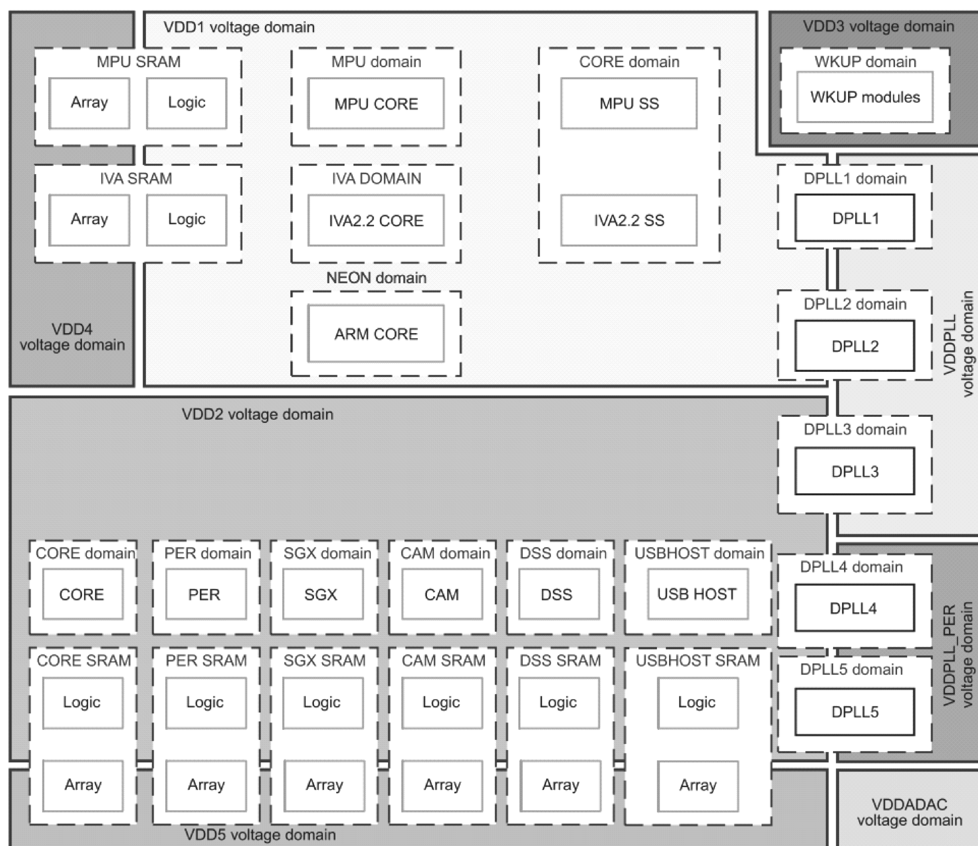


图 2-5 voltage domain 和 power domain 关系框图

时钟本身在输入和输出中就可能有不同，设计时将 DPLL 放在两个 voltage domain 中就可以起到隔离的作用。另外这样的设计可以降低软件的复杂度，由于 VDD1 和 VDD2 这两个 voltage domain 的状态是由软件管理的，如果 DPLL 完全放入这两个 voltage domain，意味着软件要完全管理 DPLL 的状态，由于 DPLL 本身的状态多样，而且管理该模块需要很多硬件的知识和背景，这样由软件管理的复杂度就比较高。如果设计中将其分离，一部分 DPLL 的开关逻辑由硬件实现，另外一部分的时钟和所连接的模块在一个 voltage domain 中，这样硬件中可以自动处理一些电源管理的功能，以简化软件设计，这也是综合考虑软件和硬件的结果。可见要设计好的系统不仅要考虑硬件的实现还要考虑软件如何控制，如果软件复杂度过高则说明系统的耦合度高，需要做硬件的简化隔离来降低整个系统的复杂度，同样在板级设计过程中也要考虑软件系统设计复杂度的问题，毕竟在嵌入式系统中最终的体验和稳定性是软件和硬件整体的结果。

看了图 2-5 之后，读者也许会有疑问，既然 power domain 的电源开关是可以控制的，那么这个控制模块是什么？又在哪个 power domain 中呢？其实图 2-4 已经给出了基本的原理，所有 power domain 的控制都是由 PRM 模块执行的，PRM、voltage domain 和 power domain 由 power Ctrl、idle/wakeup control 和 reset ctrl 三个控制信号连接。power ctrl 信号控制 voltage domain 的电源输出逻辑，决定是否为对应的 power domain 输出电源；idle/wakeup control 让 power domain 中的模块进入 idle 模式或者唤醒系统；reset ctrl 使模块进入 reset 状态。从

图 2-4 可以看到 PRM 在 WKUP domain 中。这个很好理解，这里使用反证法证明，如果 PRM 不在 WKUP domain 这个唯一保持清醒的 power domain 中，而是在某个基本 power domain 中并可以进入待机状态，那么当系统需要被唤醒的时候，就需要一个模块唤醒 PRM 所在的 power domain，这就和 PRM 管理所有基本 power domain 这种设计思想相矛盾，所以 PRM 必然在 WKUP power domain 中。各个 power domain、PRM 和 WKUP domain 之间的详细关系如图 2-6 所示。

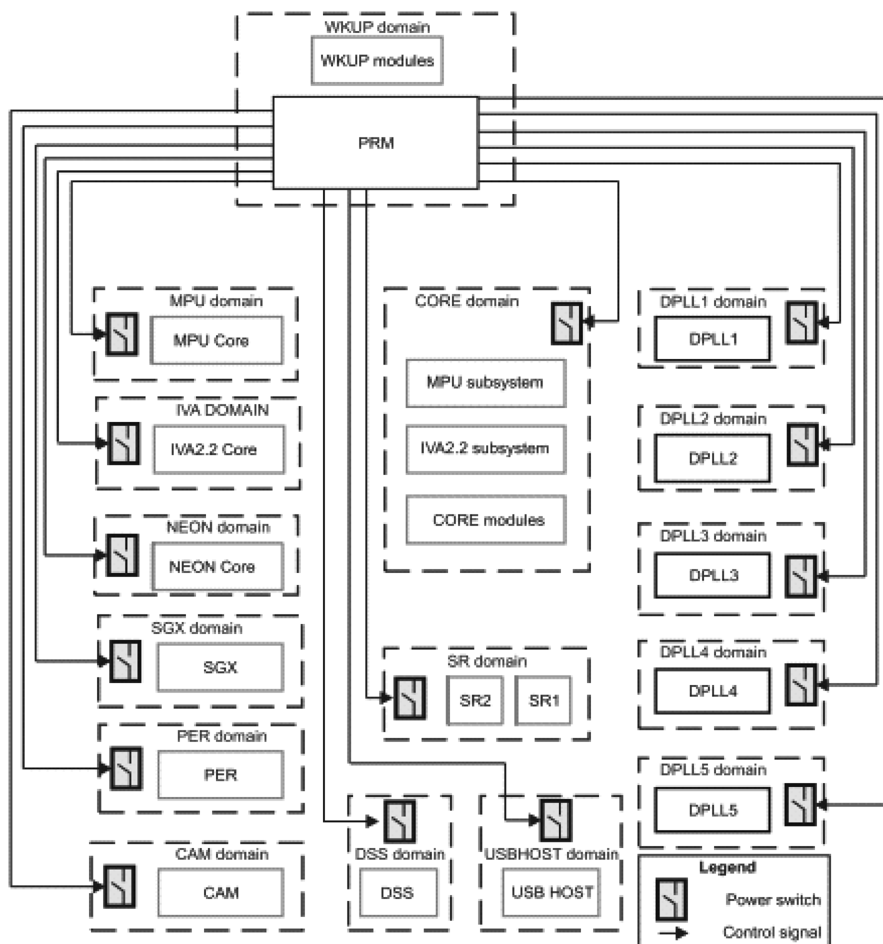


图 2-6　各个 power domain 详细关系框图

图 2-6 引自 TI 的《DM 3730 芯片手册》中第 280 页框图，从中可以看到所有的 power domain 以及 WKUP domain 中的 PRM 和这些 power domain 的控制关系。

（3）引脚唤醒

当系统进入待机状态后需要能从外部唤醒系统。这就需要 DM 3730 中的部分外部信号能在系统进入待机状态时监测外部信号的变化。之前说明中提到，WKUP domain 中有一组 GPIO 可以作为外部的唤醒源来实现唤醒系统的功能，但是只有一组 GPIO 是无法满足需求的，终端开发厂商希望外部接口同样有唤醒系统的功能。如何实现呢，笔者认为 DM 3730 中采用了比较精巧的方法，其基本原理的框架如图 2-7 所示。

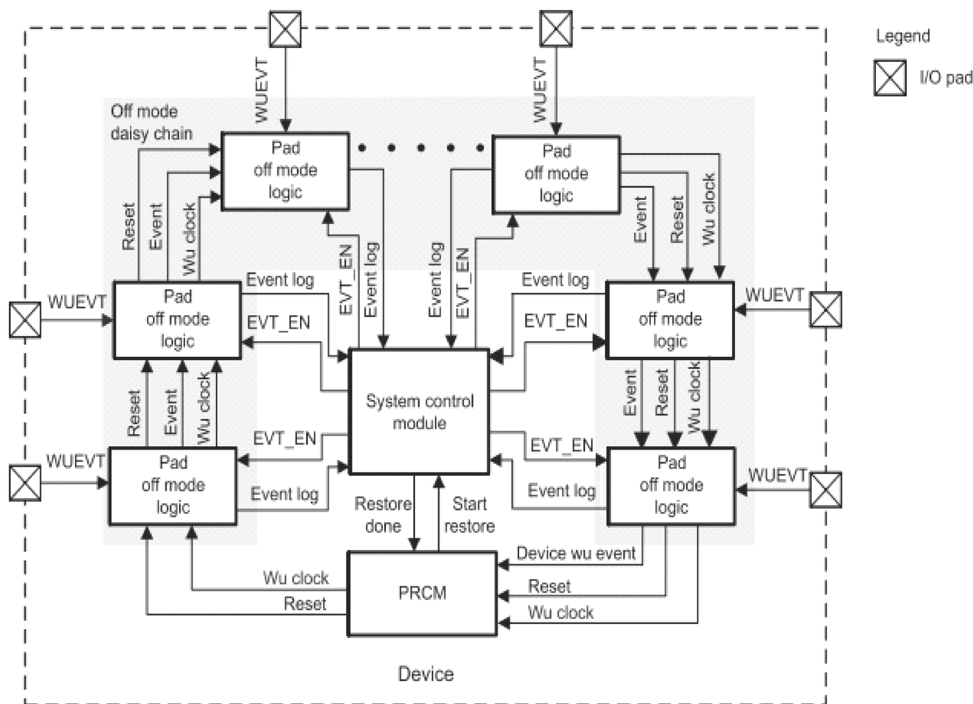


图 2-7　DM 3730 待机管理基本原理框图

图 2-7 引自 TI 的《DM 3730 芯片手册》中第 391 页框图。当系统进入待机 off 模式时，引脚的工作模式通过 pad off mode logic 来接管执行，如果已经通过 System control module（主要是控制引脚功能的模块）使能待机唤醒事件上报的功能，则该引脚会检测唤醒事件，一旦有相应的事件发生信号就会通过 pad 形成的菊花链通知一直清醒的模块（PRCM 中的 PRM），从而唤醒系统，系统被唤醒后再由相应的驱动程序检查是否由事件唤醒系统，如果是，则要进行后续的操作。可见这里硬件提供了唤醒系统和查询状态的机制，具体实现则还是依赖软件完成，所以不能孤立地看硬件实现，还是要关注软件如何进行操作，提供一个方便简单的操作逻辑对系统的实现以及其稳定性都至关重要。

（4）时钟管理

至此，PRCM 中 PRM 相关的功能和实现方法已经做了基本介绍，在 PRCM 中还有一部分就是 CM（Clock Management，时钟管理），这部分主要负责各个模块的时钟产生，整个 DM 3730 的时钟树框图如图 2-8 所示。注意图 2-8 是示意图，从中基本可以理解时钟树的层次，具体的实现有更多的细节，但是层次的概念是相同的。从图 2-8 中可以看到两个 voltage domain 分别是 wakeup voltage domain 和 core power domain，clock management 是在 core voltage domain 中，为什么 CM 不放在 wakeup voltage domain 中呢？芯片设计人员这样设计还是为了能够降低待机功耗，毕竟管理所有 clock 的模块复杂度不会太低，而且在待机状态时外设模块都应该关掉，这样把 CM 放入可以关掉的 core voltage domain 中也是理所当然的。当然 wakeup voltage domain 中的 PRM 要执行，同样需要时钟，这样设计时就将外部的晶振或者外部提供的时钟直接接入 PRM，保证这部分一直可以执行。另外图 2-8 中可以看到 CM 的输出有很多的时钟，而且每个输出时钟都有开关可以控制，这些时钟都接到了具体的模

块，也就是说每个模块的输入时钟都可以控制其开关，这样就可以通过软件控制设备的时钟输入，从而达到较好的电源管理效果。时钟管理自然也是电源管理框架中的重要一环，时钟作为设备运行的必需资源需要按需开关，按需开关时钟在设备驱动程序中是非常重要的操作。

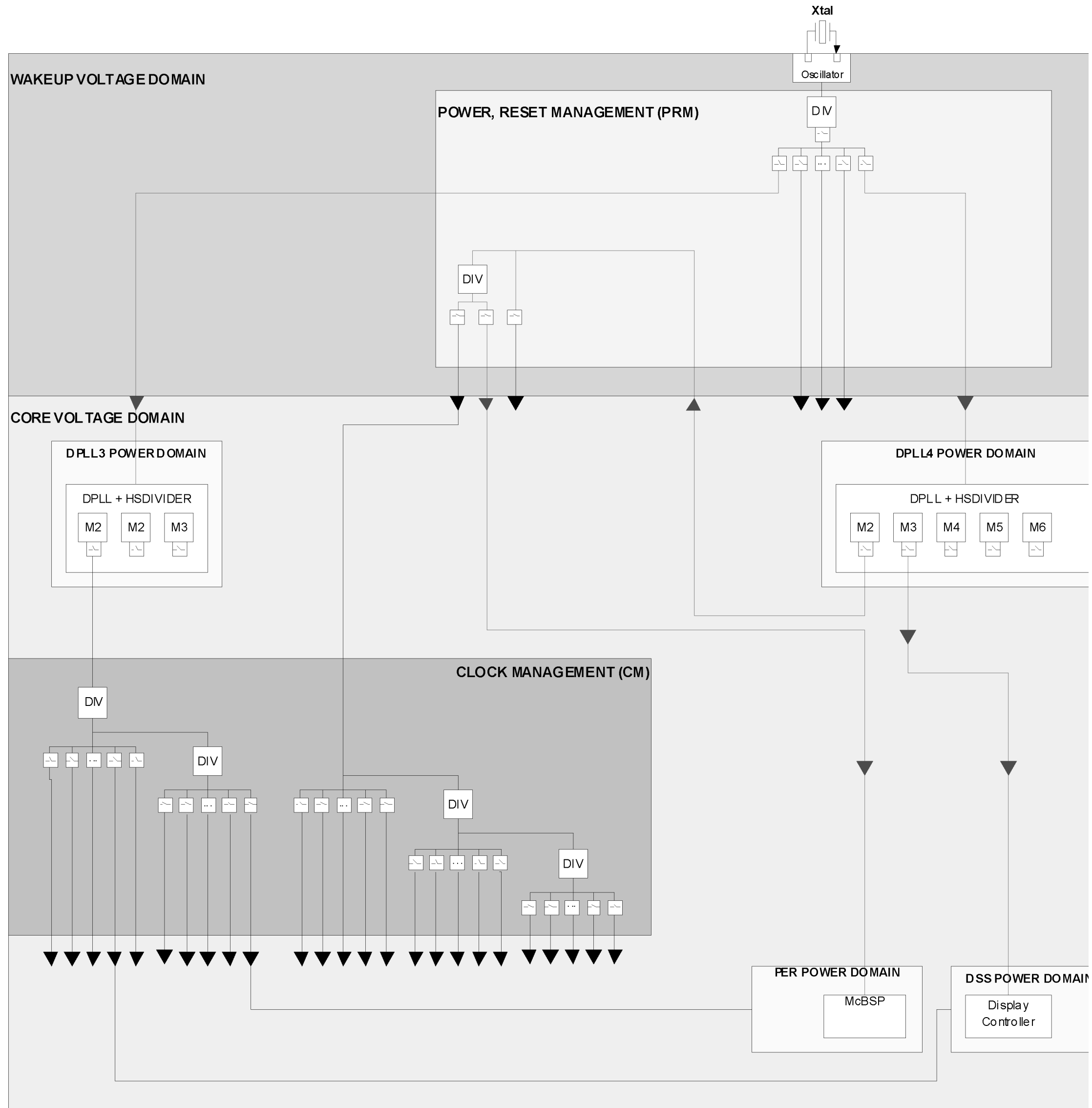


图 2-8　时钟树示意框图

具体的设备模块中通常会有两种时钟，一种是 interface clock（接口时钟，通常用于访问寄存器），另一种是 function clock（功能时钟，主要是模块运行需要的时钟）。这两种时钟通常可以单独控制，设计两种时钟是因为寄存器通常对模块进行设置或者获取其状态，而不同设备的功能时钟频率是不同的，如果使用同一种时钟，那么寄存器操作这部分就要针对每个设备模块单独设计，增加系统复杂度。

至此，DM 3730 电源管理相关设计介绍完毕，可见好的电源管理方案设计还是很复杂的，要考虑方方面面的问题。

2.2 DM 81XX 系列微处理器

2010 年 TI 针对 DaVinci 品牌系列芯片推出了 DM 81XX 系列芯片，包括针对高性能的 DM 816X 系列以及有好的性能功耗比的 DM 814X 系列。

DM 8168 视频 SoC 可提供业界最佳的视频处理性能，同时提高了多通道密度，并可支持更高的分辨率。此外，该 SoC 可显著提高高清视频的预处理与后处理功能，实现前所未有的视频性能，从而能够以更低的比特率支持更高质量的视频，满足视频安全与视频通信应用的需求。相对于其他数字视频处理器而言，该 SoC 支持高级视频分析与增强 2D 与 3D 图形的能力，DM 8168 视频 SoC 能够单片实现十六通道 H.264 等多视频格式 DVR 功能，从而可显著降低 DVR 系统的成本与复杂性。

DM 8148 视频 SoC 则可以在 3W 功耗下实现 1080P@60fps 的视频编码能力，从而达到较好的性能功耗比，适合汽车多媒体以及交通和安防相机等应用。

DM 81XX 系列芯片的应用范围和主要处理器见表 2-2。

表 2-2 DM 81XX 系列芯片应用范围和主要处理单元

芯片型号	应 用 范 围	DSP	ARM Cortex - A8	3D 加速	视 频 加 速	TI 提供编解码算法
DM 8168	安防，瘦客户机，视频会议，可视电话	1	1	1	3 HDVICPs	H.264 BP/MP/HP，MPEG-4，MPEG-2，JPEG/MJPEG
DM 8167	安防，视频服务器	1	1		3 HDVICPs	H.264 BP/MP/HP，MPEG-4，MPEG-2，JPEG/MJPEG
DM 8165	安防，视频服务器	1	1		2 HDVICPs	H.264 BP/MP/HP，MPEG-4，MPEG-2，JPEG/MJPEG
DM 8148	安防，瘦客户机，视频会议，可视电话	1	1	1	1 HDVICP	H.264 BP/MP/HP，MPEG-4，MPEG-2，JPEG/MJPEG
DM 8147	安防	1	1		1 HDVICP	H.264 BP/MP/HP，MPEG-4，MPEG-2，JPEG/MJPEG

DM 81XX 主要还是面向视频的应用。除了以上主要的应用范围，DM 81XX 还可应用在广电领域如视频转码服务器等产品上。另外由于 DM 81XX 可以提供丰富的视频输出接口，也有厂商使用该系列芯片实现如视频矩阵的特殊视频应用。总之，DM 81XX 系列芯片可以说是视频 SoC 的王者。

2.2.1 DM 81XX 系列微处理器框架

DM 81XX 处理器框架有 DM 816X 系列和 DM 814X 系列。

DM 816X 系列处理器框架如图 2-9 所示。图 2-9 引自《DM 8168 芯片数据手册》第 5 页框图。

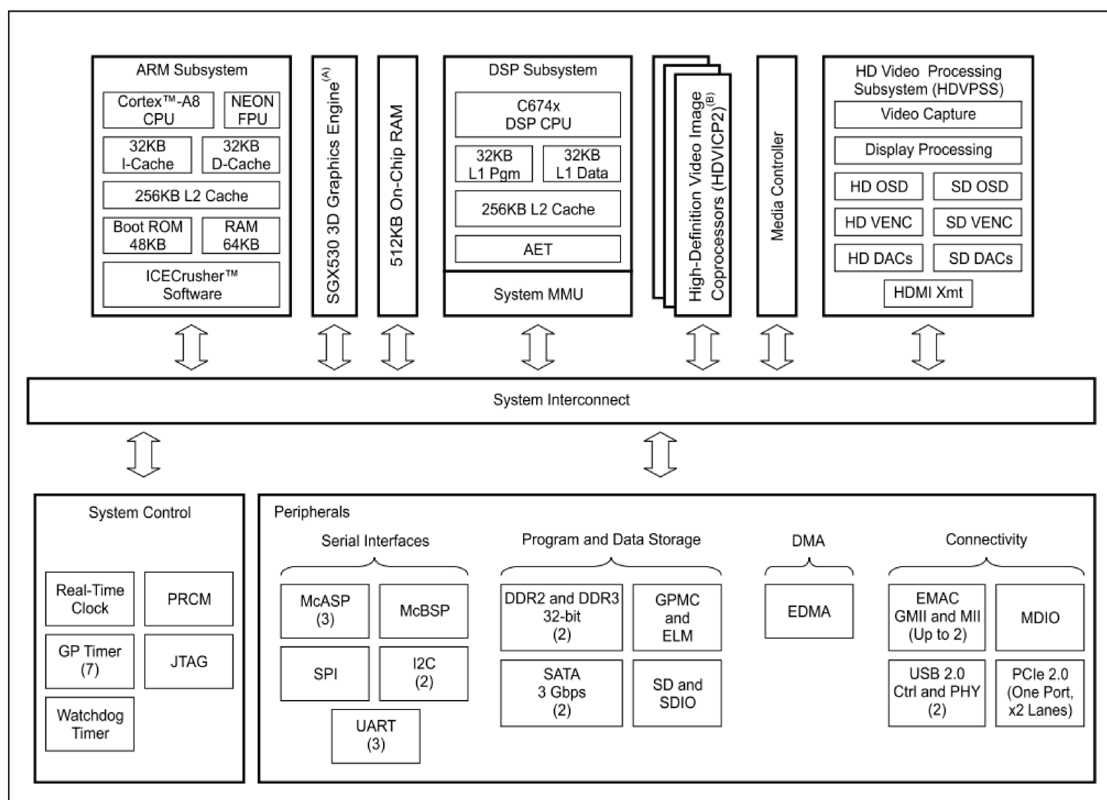


图 2-9　DM 816X 系列处理器框图

DM 816X 有六种核心处理单元：

① ARM Cortex – A8：负责系统控制和外围接口驱动。Linux 运行在该处理器上。

② DSP ：负责信号处理，可以实现各种算法如图像识别等。

③ SGX530：负责 3D 渲染的硬件加速单元（只有 DM 8168 有）。

④ HDVICP2：负责视频编解码处理，根据芯片型号会有 2、3 个 HDVICP2 （DM 8168 和 DM 8167 有 3 个），1 个 HDVICP2 有 1080P@60fps 的视频编码或解码能力。

⑤ HDVPSS：负责处理视频的输入和输出，可以处理多路复合视频输入，如 4/8/16 路标清输入，显示输出最多可以有 4 路同源的输出。内部还有很强的降噪、缩放、去隔行等功能。

⑥ Media Controller：芯片中包含两个 ARM Cortex – M3 （作为 Media Controller），实现对 HDVICP2 以及 HDVPSS 的控制和管理，主要是为了解决前一代 DaVinci 系列芯片 ARM 或者 DSP 在做视频处理时负载高的问题。有了 Media Controller，整个芯片在做视频编/解码和视频输入输出时不需要 ARM 和 DSP 的干预。

主要外围设备的接口如下：

① EMAC ：负责以太网连接，最多支持两个千兆以太网。

② PCIe 2.0 ：负责 PCI 总线连接，可以连接 PCI 设备，也可以作为 PCI 扩展设备与 X86 处理器连接。

③ SATA ：负责 SATA 接口处理，通常用来与 SATA 硬盘连接。

④ USB 2.0 Ctrl and PHY：负责 USB 的连接，此接口为 USB OTG 接口。

⑤ DDR2 and DDR3：负责 DDR memory 的连接，芯片包含两个 32 bit DDR 接口，可以提高整体的 DDR 吞吐量。

⑥ GPMC and ELM：负责各种 Flash 存储器的连接，该接口可以通过时序编程与 FPGA 进行连接以扩展功能。

⑦ Peripherals：各种外设接口如 UART（串口）、I^2C（通常负责传感器的控制）、McASP 和 McBSP（负责音频数据传输）、SD 和 SDIO（负责 SD 卡或者 WIFI 芯片的连接）等。

⑧ GP Timer：可编程的定时器，向系统提供定时功能。

除去以上还有以下模块：

① EDMA ：负责数据在外设和内存或者内存不同区块之间的复制，用来降低处理器的负载以提高系统性能。

② On－Chip RAM ：片内 RAM，使系统在外部内存没有初始化好的情况下仍可运行。通常该段内存可用于系统初始化、系统加速等。注意 DM 816X 系列芯片内部有 512KB 片内 RAM，这么大的容量可以放很多内容，比如在系统初始化的时候很多芯片由于片内 RAM 的容量限制需要 u-boot 作为二级引导程序，而 DM 816X 则可以将 u-boot 直接作为一级引导程序，从而减少系统启动时间，相应的 u-boot 需要初始化 DDR 控制器。

③ System Interconnect ：片内总线，负责在片内各个模块间建立通道。

DM 814X 系列处理器框架如图 2-10 所示。图 2-10 引自《DM 8148 芯片数据手册》第 5 页框图。

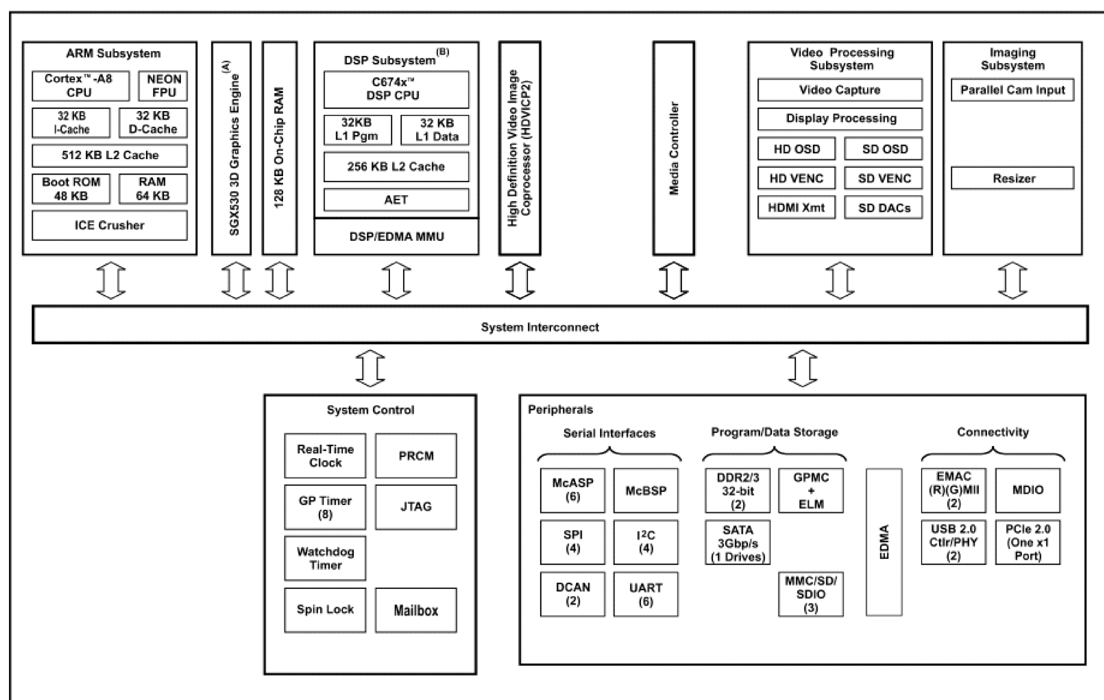


图 2-10　DM 814X 系列处理器框图

DM 814X 有七种核心处理单元：

① ARM Cortex – A8：负责系统控制和外围接口驱动。Linux 运行在该处理器上。

② DSP ：负责信号处理，可以实现各种算法如图像识别等。

③ SGX530：负责 3D 渲染的硬件加速单元（只有 DM 8148 有）。

④ HDVICP2：负责视频编解码处理，1 个 HDVICP2 有 1080P@ 60fps 的视频编码或解码能力。

⑤ Video Processing Subsystem（VPSS）：负责处理视频的输入和输出，可以处理多路复合视频输入，如 4/8/16 路标清输入，显示输出最多可以有 4 路同源的输出。内部还有很强的降噪、缩放、去隔行等功能。

⑥ Imaging Subsystem（ISS）：负责 Camera Sensor 的接入，可以接入 Bayer Patter Raw 格式数据，其中包含 ISP（Image Sensor Process）模块，可以进行图像处理，另外 ISP 中包含 3A（Auto Focus，Auto White Balance，Auto Expose）统计模块，通过 3A 统计进行图像参数的调整以达到最佳的图像效果。

⑦ Media Controller：其芯片中包含两个 ARM Cortex – M3（作为 Media Controller），实现对 HDVICP2 以及 VPSS 和 ISS 的控制和管理。

主要外围设备的接口如下：

① EMAC：负责以太网连接。

② PCIe 2.0：负责 PCI 总线连接，可以连接 PCI 设备也可以作为 PCI 扩展设备与 X86 处理器连接。

③ SATA ：负责 SATA 接口处理，通常用来与 SATA 硬盘连接。

④ USB 2.0 Ctrl and PHY：负责 USB 的连接，此接口为 USB OTG 接口。

⑤ DDR2 and DDR3：负责 DDR memory 的连接，芯片包含两个 32 bit DDR 接口，可以提高整体的 DDR 吞吐量。

⑥ GPMC and ELM：负责各种 Flash 存储器的连接，该接口可以通过时序编程与 FPGA 进行连接以扩展功能。

⑦ DCAN：负责 CAN 总线连接，CAN 总线扩展适合工业和汽车应用。

⑧ Peripherals：各种外设接口如 UART（串口）、I^2C（通常负责传感器的控制）、McASP 和 McBSP（负责音频数据传输）、SD 和 SDIO（负责 SD 卡或者 WIFI 芯片的连接）等。

⑨ GP Timer：可编程的定时器，向系统提供定时功能。

除去以上还具有同 DM 816X 相关部分相同模块。

2.2.2 DM 81XX 系列微处理器特性

DM 816X 系列处理器主要特性：

- 对混合型安防 DVR 解决方案而言，可同时支持 16 通道 D1 的 H.264 HP 编码并附带 CIF 子码流的编码与 8 通道 D1 解码，并具有视频混合与图像混合功能，支持多达三个独立显示器。
- 对视频通信应用而言，可同时支持三个 1080P@ 60fps 视频编码或解码，由于编解码器延时低于 50 ms，因此片外失效问题得以消除，从而可将点对点延时降至 50 ms

以下。

- 极高的集成度，集成 1 GHz ARM Cortex－A8、1 GHz TI C674x DSP、两个可编程高清视频影像协处理器、一个创新型高清视频处理子系统以及综合编解码器（支持包括高清分辨率的 H.264、MPEG－4 以及 VC1）。
- 包括千兆以太网、PCI Express、SATA2、DDR2、DDR3、USB 2.0、MMC/SD、HDMI 以及 DVI 等多种接口，可支持高度灵活的设计方案实施。
- 无缝接口连接至四个 TI 的多通道视频解码器 TVP5158，可无缝捕获多达十六个 D1 视频通道。TVP5158 可自动控制对比度，降低噪声，提高压缩比与整体视频质量，从而不但可取消额外的 FPGA 与外部存储器，还可简化设计，提高系统灵活性。
- 提供 1.8 V/3.3 V I/O 电压，适合工业应用。

DM 814X 系列处理器主要特性基本同 DM 816X，编解码性能方面为一个 1080P@60fps 视频编码或解码，附加的特性如下：

- 内部实现 ISP 可以实现图像色彩还原、图像增强、视频稳定以及变焦失真校正等影像信号处理技术。
- 可实现高级运动补偿低照技术，可以在极低光线环境下实现清晰的影像画质。
- 集成 CAN 总线控制器，可以扩展到汽车上的图像应用。

2.2.3 DM 81XX 系列微处理器电源管理相关设计

1. DM 81XX 使用的电源管理技术介绍

在电源管理设计方面，DM 816X 和 DM 814X 也是通过 PRCM 模块实现电源管理功能的，但是 DM 816X 和 DM 814X 中 PRCM 使用的电源管理技术是不同的。在讲 DM 3730 电源管理时提到过两种基本的电源管理技术 DVFS 和 AVS，由于 DM 816X 更多面向高性能的设备需求，所以在电源管理技术中选择了 AVS，而 DM 814X 的需求是获得最大的能耗比，所以在电源管理技术方面选择了 DVFS。而对完全待机的模式（off 模式），两个系列芯片都省略了该功能，以降低 PRCM 的复杂度。可见电源管理技术的使用还是要根据需求进行选择和实施，这样才能设计出有良好性价比的芯片。

2. DM 81XX 电源管理模块详解

电源管理 PRCM 中 voltage domain 和 power domain 的实现，在 DM 816X 和 DM 814X 两个系列芯片中的设计也是不同的。

DM 816X 中 voltage domain 和 power domain 的关系如图 2-11 所示。图 2-11 引自《DM 8168 芯片手册》中第 1788 页框图。框图中下半部分的表格说明图中各个颜色属于哪个 voltage domain 和 power domain，其中列表示的是 voltage domain，而行表示的是 power domain。从框图中可以看到不同模块的逻辑部分和存储部分还是采用不同的电压。在 voltage domain 中有一个是 1 V AVS，这个 1 V AVS voltage domain 就是指使用 AVS 技术，其电压规格是 1 V 的 voltage domain。框图中主要的处理模块（如 ARM、DSP、HDVPSS 和 HDVICP 等）和主要的外部接口（如 PCIe、SATA 等）都在 1V AVS voltage domain 中，可以说整个芯片的逻辑部分都在 1V AVS 的管理下，这样设计可以最大限度地降低整个芯片的功耗，在技术和实际效果中取得一个好的平衡。从 power domain 来看 DM 816X，可以看到有个 Always－on power domain，该 power domain 会在上电后电源处于常开状态，Always－on power domain 中包括

ARM 和大部分外设接口，从芯片设计的角度是简化了许多。

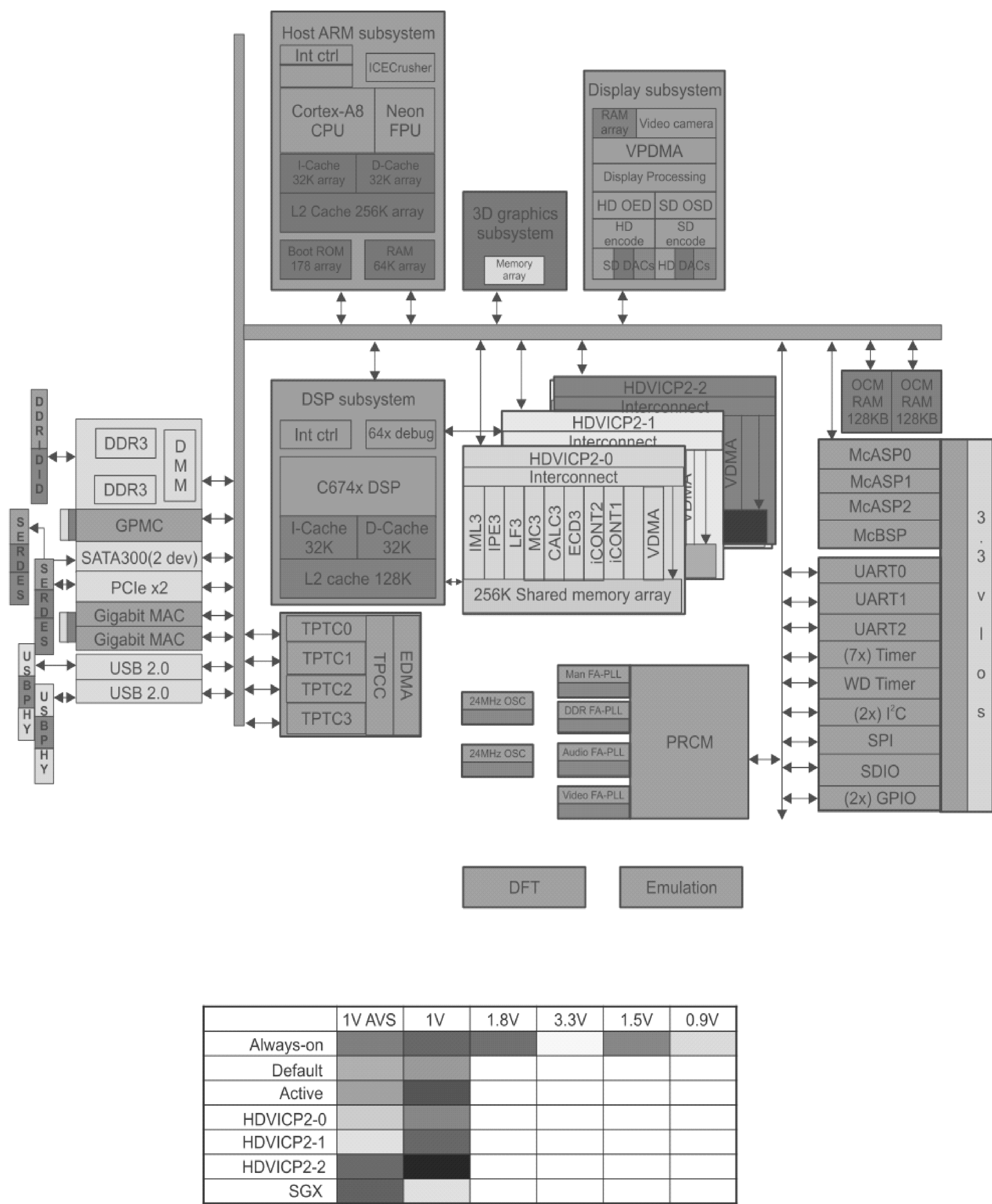


图 2-11　DM 816X 的 voltage domain 和 power domain 关系框图

下面看看 DM 814X 中 voltage domain 和 power domain 的关系，如图 2-12 所示。

图 2-12 引自《DM 8148 芯片手册》中第 487 页框图。从中可以看出 DM 814X 的 voltage domain 和 power domain 的关系，power domain 中逻辑相关的部分在一个 voltage domain 中，而 memory 存储相关的部分在另外一个 voltage domain 中。DM 814X 的 power domain 信息以及其中包含的具体模块见表 2-3。

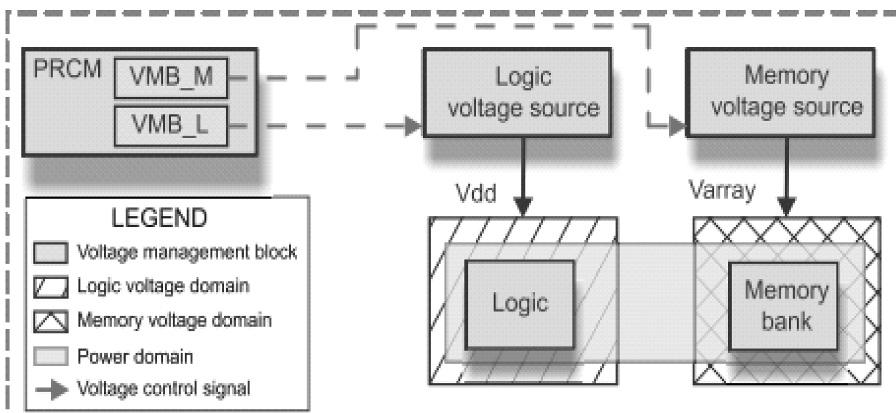


图 2-12　DM 814X 的 voltage domain 和 power domain 的关系框图

表 2-3　DM 814X Power Domain 信息及模块说明

电　源　域	模　　块
ALWAYS_ON	Cortex™ - A8，DCAN0，DCAN1，DDR_PHY_LOGIC0，DDR_PHY_LOGIC1，DMM EDMA，ELM，EMIF0，EMIF1，2Channel GMAC Switch，GPIO_CNTL0，GPIO_CNTL1，GPIO_CNTL2，GPIO_CNTL3，GPMC，I2C0，I2C1，I2C2，I2C3，Interconnect，IPC，MCASP0，MCASP1，MCASP2，MCASP3，MCASP4，MCASP5，ATL，MCBSP，MLB，OCMC SRAM，PATA，PBIST，PCI，PRCM，RTC，SATA，SCR，SD/MMC0，SD/MMC1，SD/MMC2，SPI0，SPI1，SPI2，SPI3，Switch fabric，Timer1，Timer2，Timer3，Timer4，Timer5，Timer6，Timer7，Timer8，UART0，UART1，UART2，UART3，UART4，UART5，USB0，USB1，WDT0，VCP，XBAR，Debug_SS
ISS（ISP）	FACEDETECT，ISS
SGX530	SGX530 Subsystem
DSS	HDD_SS，HDMI，SD - DAC LOGC
ACTIVE（DSP）	C674x + ™Subsystem. ，C674x + ™L2 SRAM
HDVICP	HDVICP，SRSense2

DM 816X 和 DM 814X 的内部时钟设计和 DM 3730 相比也做了精简。有很多模块共用同一个时钟，而不像 DM 3730 那样每个模块都有自己的时钟。

DM 816X 系统内部时钟信息见表 2-4。

DM 816X 的外部主要时钟输入是 27 MHz，但是从表中可见 SYSCLK18 是 32 kHz，这是因为 DM 816X 中包含一个 RTC，这个 32 kHz 是为 RTC 提供的，而不是像 DM 3730 中为整个系统待机时使用，同样的时钟源其使用方法也会因为需求的变化而不同。

DM 814X 的时钟框架如图 2-13 所示。从中可以看出时钟框架同样做了类似 DM 816X 的精简，一个系统时钟会对应多个模块。其中也有奇怪的现象，比如有 OSC0 和 OSC1 两个时钟源，而且 OSC0 是 20 MHz 固定的，而 OSC1 是 20 ~ 30 MHz 之间的，这样设计是为了音频和视频的精度，比如音频通常是 44. 1 kHz 采样频率。通过两个时钟源就可以提高音频采样的精度，提高音频质量，也就从整体上提升产品质量。

表 2-4　DM 816X 内部时钟信息

时　钟	频　率	描　述
SYSCLK1	1 GHz	To C674x DSP
SYSCLK2	1. 2 GHz	To Cortex – A8
SYSCLK3	600 MHz	To HDVICP2
SYSCLK4	500 MHz	Interconnect Clock, Clock for HD DSS, TPTCs, TPCC, DMM
SYSCLK5	250 MHz	Interconnect Clock, SGX530, USB SS, 10/100/1000 EMAC, SATA, PCIe, OCMC RAM
SYSCLK6	125 MHz	Interconnect Clock, UART, I^2C, SPI, SDIO, TIMER, GPIO, McASP, McBSP, GPMC
SYSCLK7	125 MHz(maximum)	Reserved
SYSCLK8	800 MHz	DDR clock
SYSCLK9	48 MHz	Reserved
SYSCLK10	48 MHz	SPI. I^2C, SDIO, and UART functional clock
SYSCLK11	216 MHz	Reserved
SYSCLK13	165 MHz(maximum)	HDVPSS
SYSCLK14	27 MHz	Reserved
SYSCLK15	165 MHz(maximum)	HDVPSS
SYSCLK16	27 MHz	Reserved
SYSCLK17	54 MHz	HDVPSS
SYSCLK18	32 kHz	RTC
SYSCLK19	62. 5 MHz	Reserved
MCASP0_CLK		McASP0 AUX Clock
MCASP1_CLK		McASP1 AUX Clock
MCASP2_CLK		McASP2 AUX Clock
MCBSP_CLK		McBSP AUX Clock
TIMER1_CLK		Timer clock for Timer 1
TIMER2_CLK		Timer clock for Timer 2
TIMER3_CLK		Timer clock for Timer 3
TIMER4_CLK		Timer clock for Timer 4
TIMER5_CLK		Timer clock for Timer 5
TIMER6_CLK		Timer clock for Timer 6
TIMER7_CLK		Timer clock for Timer 7

至此 DM 81XX 电源管理相关设计就介绍完毕，可见设计要从需求出发，考虑最佳性价比。

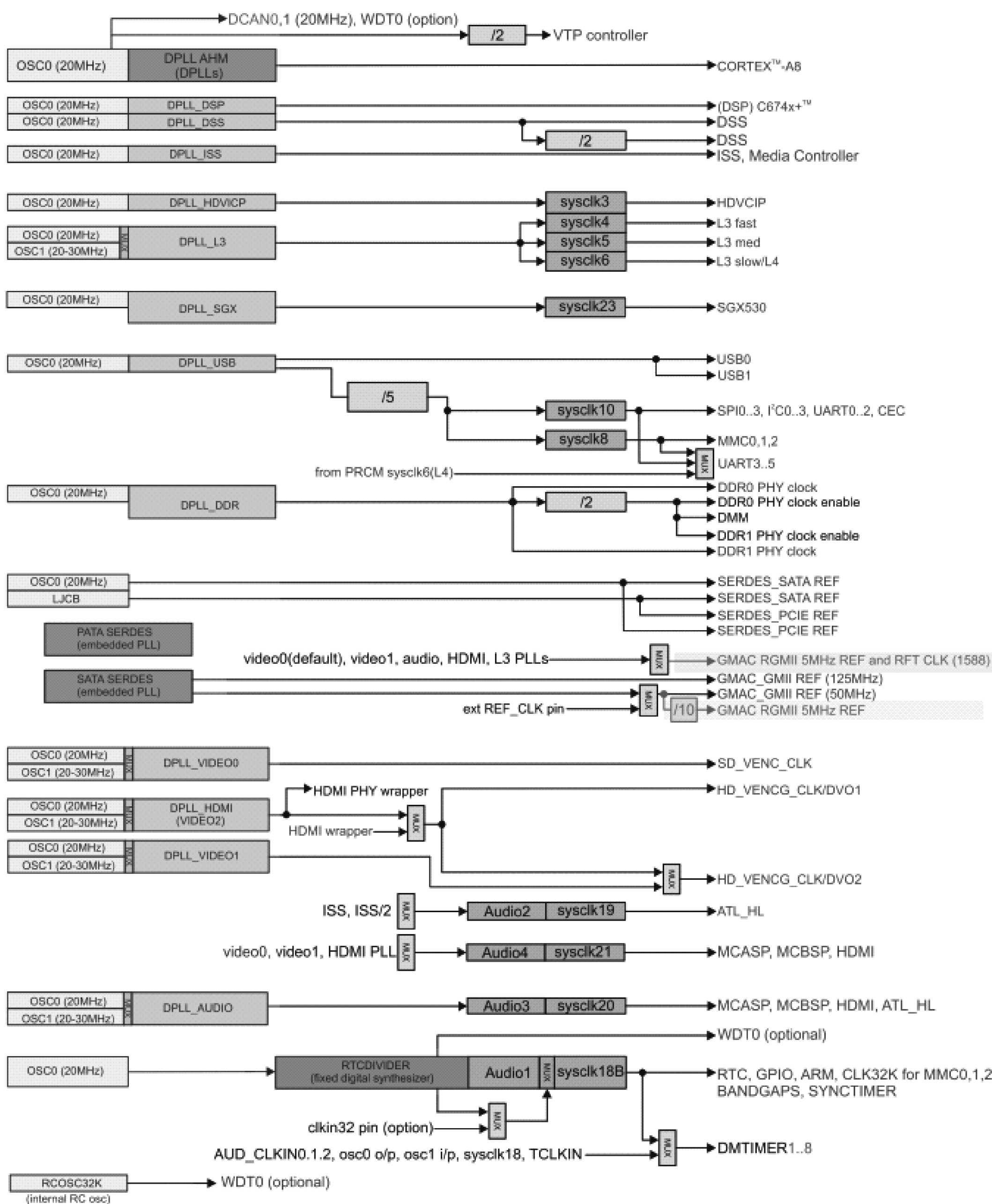


图 2-13　DM 814X 时钟框架

2.3　Sitara 系列芯片

笔者看到一篇很好的关于处理器的文章是王齐的《ARM 与 X86》。在这里介绍给读者，其中可以了解到 ARM 和 X86 两种不同架构的发展史，在 ARM 的发展中 TI 一直是幕后英雄，ARM 真正的发展是依赖于 TI 与 Nokia 合作的无线终端芯片的发展。但是 2009 年之前 TI 对 ARM 的使用都是在无线终端和视频芯片范围内，并没有像三星、ST 等推出纯 ARM 的产品。

这个格局在2010年被打破了，面对广阔的嵌入式控制市场，TI在2010年之后推出了一系列纯ARM的处理器产品即Sitara系列微处理器。既然TI已经在无线终端和视频芯片中使用了ARM，那么最简单的推出纯ARM处理器的方法就是在无线终端和视频芯片的基础上精简，去除不使用的协处理器。这样做的好处很明显，没有开发的费用，而且和相应的芯片引脚兼容，方便嵌入式设备开发商在一个平台开发多种类型产品。具体Sitara系列芯片应用说明见表2-5。

表2-5　Sitara系列芯片应用说明

芯片型号	应用范围	ARM Cortex－A8	3D加速	I/O Supply(V)
AM 3894	通信和电信，计算机及外设，工业，医疗	1	1	1.5，1.8，3.3
AM 3892	通信和电信，计算机及外设，工业，医疗	1		1.5，1.8，3.3
AM 3874	通信和电信，计算机及外设，工业，医疗	1	1	1.5，1.8，3.3
AM 3871	通信和电信，计算机及外设，工业，医疗	1		1.5，1.8，3.3
AM 3715	消费性电子，工业，医疗	1	1	1.8
AM 3703	消费性电子，工业，医疗	1		1.8
AM 3517	消费性电子，工业，医疗	1	1	1.8，3.3
AM 3505	消费性电子，工业，医疗	1		1.8，3.3
AM 3359	连接自动贩卖机，家庭/楼宇自动化，消费类电子产品	1	1	1.8，3.3
AM 3358	便携式导航，连接自动贩卖机，家庭/楼宇自动化，消费类电子产品	1	1	1.8，3.3
AM 3357	便携式导航，连接自动贩卖机，家庭/楼宇自动化，消费类电子产品	1		1.8，3.3
AM 3356	连接自动贩卖机，家庭/楼宇自动化，消费类电子产品	1		1.8，3.3
AM 3354	便携式导航，连接自动贩卖机，家庭/楼宇自动化，消费类电子产品	1	1	1.8，3.3
AM 3352	连接自动贩卖机，家庭/楼宇自动化，消费类电子产品	1		1.8，3.3

表2-5中AM 3894和AM 3892是从DM 816X系列芯片精简得来的；AM 3874和AM 3871是从DM 814X系列芯片精简得来的；AM 3715和AM 3703是从DM 3730系列芯片精简得来的。

由于无线终端芯片和视频芯片开发周期较长，成本较高，一味的采用精简的方式并不能适应对开发周期和成本要求都比较高的消费类ARM的市场。为了适应相应的市场需求，TI于2011年推出了AM 335X系列处理器，Cortex－A8的核心最低价格低于5美元，性价比还是很高的。

2.3.1　Sitara系列芯片框架

由于部分Sitara芯片是从对应的无线终端芯片和视频芯片精简而来的，这里就重点介绍AM 335X系列处理器的框架。

AM 335X系列处理器框架如图2-14所示。图2-14引自《AM 335X芯片数据手册》第6页框图。

AM 335X有五种核心处理单元：

① ARM Cortex－A8：负责系统控制和外围接口驱动。Linux运行在该处理器上。

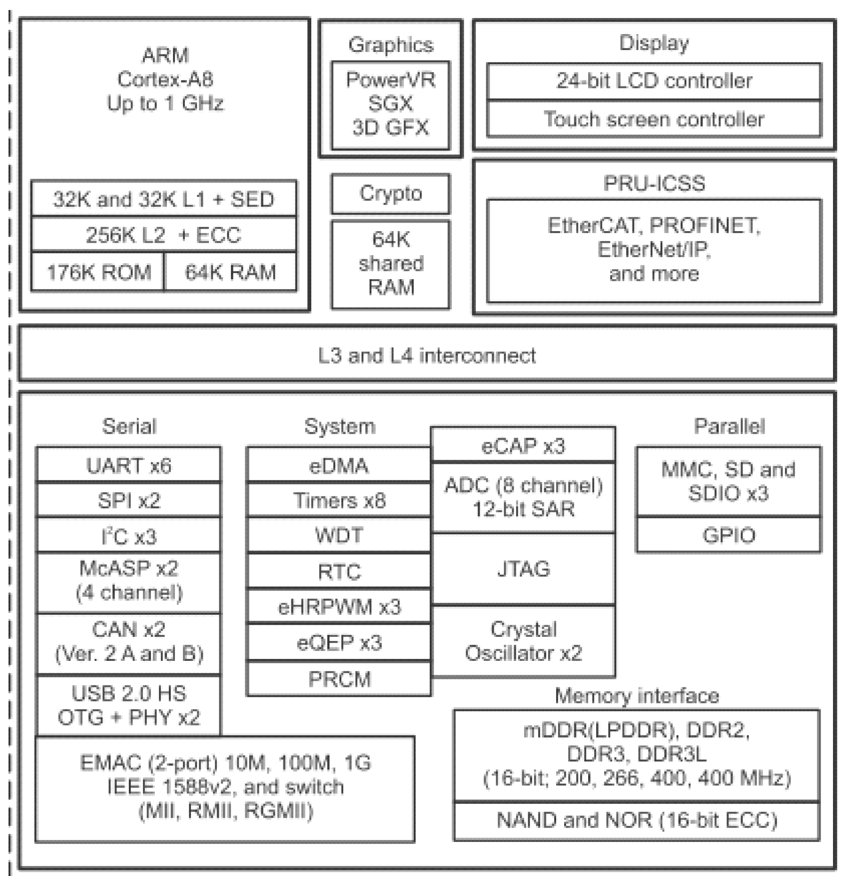


图 2-14　AM 335X 系列处理器框图

② PowerVR SGX：负责 3D 渲染的硬件加速单元。

③ Display：负责处理视频的显示输出。

④ Crypto：负责加密算法的加速处理单元。

⑤ PRU - ICSS：可编程单元实现对引脚的管理，从而可以根据需要模拟不同的接口功能，实现类似 FPGA 的功能。

主要外围设备的接口如下：

① EMAC：负责以太网连接，最多支持两个千兆以太网。

② USB 2.0HS OTG + PHY：负责 USB 的连接，此接口为 USB OTG 接口。

③ Memory Interface：负责 DDR memory 内存的连接，以及 Flash 存储器的连接。

④ Peripherals：各种外设接口如 UART（串口），I^2C（通常负责传感器的控制），McASP（负责音频数据传输），SD 和 SDIO（负责 SD 卡或者 WIFI 芯片的连接）等。

⑤ GP Timer：可编程的定时器，向系统提供定时功能。

除去以上还有以下模块：

① eDMA：负责数据在外设和内存或者内存不同区块之间的复制，用来降低处理器的负载以提高系统性能。

② 64K Shared RAM：片内 RAM，使系统在外部内存没有初始化好的情况下仍可运行。

通常该段内存可用于系统初始化、系统加速等。

③ L3 and L4 Interconnect：片内总线，负责在片内各个模块间建立通道。

2.3.2 Sitara 系列芯片特性

从对应的无线终端芯片和视频芯片精简而来的 Sitara 芯片的特性之前已经有所了解，这里重点介绍 AM 335X 系列处理器，其特性如下：

- AM 335x ARM Cortex－A8 微处理器适用于工业自动化设计，提供可编程实时单元（PRU）片上接口，可实现实时工业通信（主从），支持 EtherCAT ®、Ethernet/IP、PROFIBUS ®、PROFINET ®、POWERLINK 以及 SERCOS III 等常见协议。AM 335x ARM 微处理器中这种独特的 PRU＋ARM 架构无需外部 ASIC 或 FPGA，可降低系统复杂性，节省超过 30% 的材料清单（BOM）成本。此外，AM 335x ARM 微处理器还包含其他重要片上工业外设（CAN、ADC、USB＋PHY 以及双端口千兆以太网 IEEE1588），不但支持快速网络连接与快速数据吞吐，而且还可连接传感器、传动器以及电机控制。
- 支持众多不同终端设备的统一可扩展平台，设计人员可充分利用 ARM Cortex－A8 微处理器的引脚以及软件兼容性，采用最符合工业自动化需求的器件设计不同的终端设备。
- 芯片上提供高级 3D 图形功能、触摸屏控制器以及高级外设，为开发人员提供高性能的基础上缩减板级空间，降低设计复杂性，并可将材料清单（BOM）成本锐降 40 美元，从而充分满足便携式导航、掌上游戏及教育设备，以及家庭及楼宇自动化等更小型应用的需求。

2.3.3 Sitara 系列芯片电源管理相关设计

在电源管理方面，仍以 AM 335X 的实现为主进行介绍。

1. AM 335X 使用的电源管理技术介绍

由于 AM 335X 需要满足便携式消费电子产品的需求，所以在电源管理技术方面使用了 DVFS 和 AVS。另外 AM 335X 也允许系统进入完全待机状态，和 DM 3730 不同的是待机时 AM 335X 功耗大概 3 mW 左右。3 mW 的待机功耗虽然无法和 DM 3730 的 0.1 mW 相比，但是普通的便携设备也是可以接受的。

2. AM 335X 电源管理模块详解

AM 335X 在电源管理的实现虽然使用的技术是全面的，但是如果和 DM 3730 在 PRCM 的实现进行比较，AM 335X 的实现还是做了比较多的精简。完全待机 3 mW 功耗也能看出这种精简的结果。

在 voltage domain 实现方面，AM 335X 实现了两个 voltage domain 分别是 VDD_CORE 和 VDD_RTC。所有核心模块都放在了 VDD_CORE 中，只要将 RTC 这个需要单独供电的模块放入 VDD_RTC 中，就能保证 RTC 在其他模块下工作的情况下仍然可以工作，确保时间的准确。

在 power domain 实现方面，AM 335X 实现了多个 power domain：为了能唤醒系统提供了 WKUP power domain，ARM 单独是一个 power domain 称作 MPU power domain；RTC 在 RTC

power domain 中；大部分外设则在 PER power domain 中。和 DM 3730 一样，WKUP 还包含一些外设模块，可以用来唤醒系统，但是为了设计的简单，并未实现 WKUP domain 之外模块通过 I/O 唤醒系统，这样就又比 DM 3730 设计简单了许多。

在时钟管理方面，AM 335X 直接通过 Core PLL 分频出几个时钟供大部分的外设模块使用。Core PLL 的时钟树如图 2-15 所示，图 2-15 引自《AM 335X 芯片手册》中第 893 页框图，从图中可以看出其输出有限的几个时钟如 L3F_CLK、L3S_CLK 等，Core PLL 的时钟输出与各外设模块的对应关系见表 2-6。

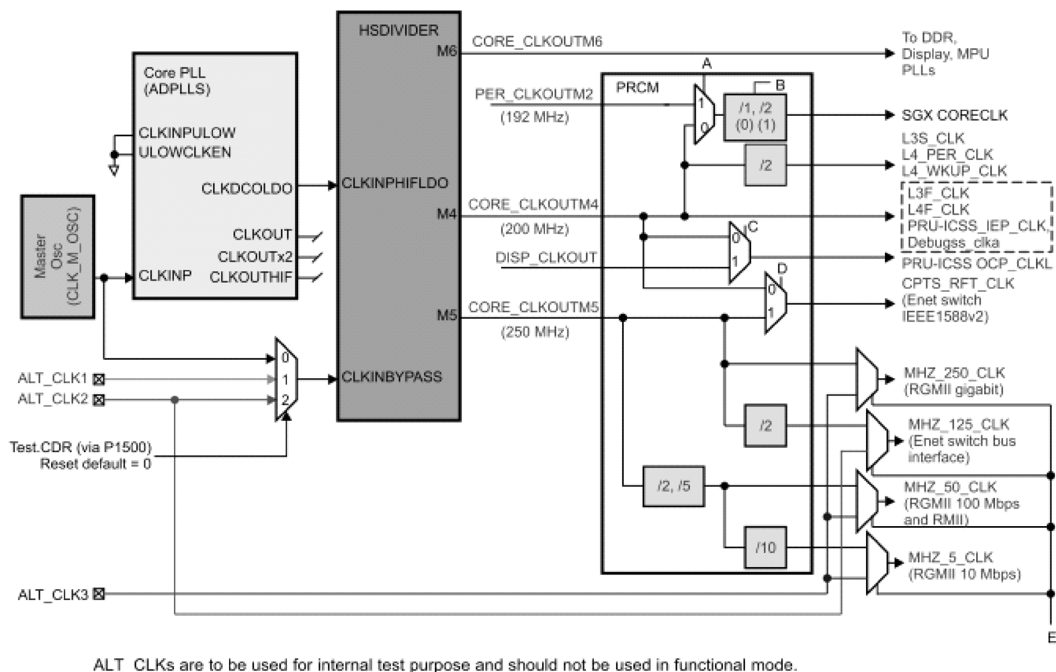


图 2-15　AM 335X Core PLL 时钟树

表 2-6　Core 时钟输出对应的设备模块

时　钟	模　块
L3F_CLK	GEMAC Switch（Ethernet），DAP，PRU－ICSS，EMIF，TPTC，TPCC，OCMC RAM，DEBUGSS
L3S_CLK	USB，TSC，GPMC，MMCHS2，McASP0，McASP1
L4_PER_CLK	DCAN0，DCAN1，DMTIMER2，DMTIMER3，DMTIMER4，DMTIMER5，DMTIMER6，DMTIMER7，eCAP/eQEP/ePWM0，eCAP/eQEP/ePWM1，eCAP/eQEP/ePWM2，eFuse，ELM，GPIO1，GPIO2，GPIO3 I2C1，I2C2，IEEE1500，LCD，Mailbox0，McASP0，McASP1，MMCHS0，MMCHS1，OCP Watch-point，SPI0，SPI1，Spinlock，UART1，UART2，UART3，UART4，UART5
L4_WKUP_CLK	ADC _ TSC，Clock Manager，Control Module，DMTIMER0，DMTIMER1 _ 1MS，GPIO0，I2C0，M3UMEM，M3DMEM，SmartReflex0，SmartReflex1，UART0，WDT1

系统中还有一些时钟（如 ARM 的时钟）也是通过类似 Core PLL 的方式由 PLL 输出的，这里就不再赘述，具体细节可见《AM 335X 芯片手册》第 8.1.6 节。

在 WKUP 的设计中，AM 335X 在待机和唤醒控制方面，通过将 Cortex－M3 放入 WKUP

domain 中，然后采用软件控制的方式实现相应的功能，没有像 DM 3730 那样直接用硬件实现复杂的控制逻辑。相对于硬件来说软件开发成本较低，将这部分用成熟核心和软件共同实现，可以提高硬件的稳定性，降低成本，虽然效果没有单独设计硬件好，但是综合下来在满足需求的情况下，该实现方案还是很有优势的。

从 AM 335X 电源管理的设计细节可以看出技术方案还是多种多样的。符合需求，性价比好才是最好的，“只选对的，不选贵的”在设计中也非常适用。

2.4 TI 处理器内核特殊代码结构

在 Linux 中，体系结构相关的代码都在 arch 目录下。之前介绍的 TI 处理器都属于 ARM 体系结构。在 arch/arm 目录下，和 TI 处理器相关的目录有 mach – davinci、mach – omap1、mach – omap2、plat – omap。这些目录相关的说明见表 2-7。

表 2-7　TI 处理器内核特殊代码目录说明

目　　录	说　　明
mach – davinci	该目录中的文件是以 ARM9 为核心的 DaVinci 处理器（如 DM 6446、DM 36X 和 DM 6467）的设备相关的底层代码
mach – omap1	该目录中的文件是第一代 OMAP 处理器的设备相关的底层代码（该目录处理器已经过时，很少使用）
mach – omap2	该目录中的文件是 OMAP2/OMAP3/OMAP4/OMAP5 处理器和新一代 DaVinci 处理器（DM 81XX 系列处理器）的设备相关的底层代码
plat – omap	该目录中的文件是为驱动抽象出来的各种与设备无关的统一接口，用以减少系统耦合度

表 2-7 中有一个奇怪的现象，新一代 DaVinci 处理器的代码不是在 mach – davinci 中，而是在 mach – omap2 中。这看似奇怪的问题，可以换个角度来考虑，既然把它们放到一起必定是有原因的，这个原因是什么呢？老话常说“物以类聚，人以群分”，软件也不例外，放到一起的代码相似度也要高一些。从这个角度考虑，新一代 DaVinci 芯片应该和 OMAP2 及以后的 OMAP 处理器在芯片内部结构方面相似度更高一些。之前介绍 DM 3730，DM 81XX 和 Sitara 系列芯片在电源管理部分都有模块 PRCM（DM 36X 等老的 DaVinci 没有 PRCM 模块），这个 PRCM 实际就是相似度的表现之一。另外芯片内的某些接口模块也是相同的。这些相似度使得将新一代 DaVinci 芯片代码放入 mach – omap2 目录下管理更合适。新一代 DaVinci 芯片的代码也一并在 OMAP3 的代码分支下管理，TI 自己维护的代码分支如下：

http://arago – project. org/git/projects/linux – omap3. git

Sitara 系列芯片的代码也是在 mach – omap2 目录下，但是代码管理在单独的分支下，TI 自己维护的 Sitara 代码分支如下：

http://arago – project. org/git/projects/linux – am33x. git

了解软件系统一个好的方式是从 Makefile 入手进行分析。下面从 linux – omap3 分支的 mach – omap2 目录下的 Makefile 入手，了解 OMAP 和 DaVinci 芯片内核特殊代码的结构。其中 OMAP 相关的内容笔者会保留和 DM 3730 相关的部分（DM 3730 属于 OMAP3 系列芯片），

去除 OMAP2 和 OMAP4 相关的内容。

```
# arch/arm/mach - omap2/Makefile
# Makefile for the linux kernel.

#适合所有芯片相关的代码,其中包括芯片 ID,I/O 地址操作,系统控制模块相关操作,平台设备相关
#操作,gp timer 相关操作,引脚复用相关操作等代码。这些代码是使用该目录所有芯片共用的
# Common support
obj - y  := id. o io. o control. o mux. o devices. o serial. o gpmc. o timer - gp. o pm. o \
            common. o gpio. o dma. o wd_timer. o elm. o
#OMAP3 中断控制器和内存控制器代码
omap - 2-3- common                                = irq. o sdrc. o

#hwmod 是抽象出来的管理实体,是硬件接口模块的抽象实体。通过该抽象将模块资源数据和操
#作分离,降低耦合度并减少冗余代码
hwmod - common                                    = omap_hwmod. o \
                                                 omap_hwmod_common_data. o
#通用的时钟管理代码
clock - common                                    = clock. o clock_common_data. o \
                                                 clkt_dpll. o clkt_clksel. o
#根据具体的芯片系列加载合适的通用模块
obj - $(CONFIG_ARCH_OMAP3) += $(omap - 2-3- common) $(hwmod - common)
obj - $(CONFIG_ARCH_TI81XX) += $(omap - 2-3- common) $(hwmod - common)

#McBSP 设备底层操作代码
obj - $(CONFIG_OMAP_MCBSP) += mcbsp. o

#和处理器相关联的电源管理芯片代码,具体根据板级使用的芯片进行配置
obj - $(CONFIG_TWL4030_CORE)                      += omap_twl. o
obj - $(CONFIG_REGULATOR_TPS65023)                += pmic_tps65023. o

#OMAP3 需要在片内 RAM 中放入在特殊情况下使用的代码。由于这些代码包括的操作会使外部
#DDR memory 的状态发生变化,只有存放在片内 RAM 才能保证稳定
# Functions loaded to SRAM
obj - $(CONFIG_ARCH_OMAP3)                        += sram34xx. o

#编译的配置参数
AFLAGS_sram34xx. o                                := - Wa, - march = armv7 - a

#与具体芯片相关的引脚复用的代码
# Pin multiplexing
obj - $(CONFIG_ARCH_OMAP3)                        += mux34xx. o
```

```
obj - $(CONFIG_ARCH_TI81XX)                       += mux81xx. o mux814x. o
#DDR 控制器相关的代码
# SMS/SDRC
obj - $(CONFIG_ARCH_OMAP2)                        += sdrc2xxx. o
# obj - $(CONFIG_ARCH_OMAP3)                      += sdrc3xxx. o
#DVFS 的各个操作点的参数
# OPP table initialization
ifeq ( $(CONFIG_PM_OPP),y)
obj - y                                           += opp. o
obj - $(CONFIG_ARCH_OMAP3)                        += opp3xxx_data. o
obj - $(CONFIG_ARCH_TI814X)                       += opp814x_data. o
endif
#与芯片相关的实现 Linux 电源管理功能的代码,其中包括实现待机,运行时 idle 和 AVS 相关的
#代码
# Power Management
ifeq ( $(CONFIG_PM),y)
obj - $(CONFIG_ARCH_OMAP3)                        += pm34xx. o sleep34xx. o sleep3517. o voltage. o\
                                                   cpuidle34xx. o pm_bus. o
obj - $(CONFIG_ARCH_TI81XX)                       += voltage. o pm_bus. o pm81xx. o sleep814x. o
obj - $(CONFIG_PM_DEBUG)                          += pm - debug. o
obj - $(CONFIG_OMAP_SMARTREFLEX)                  += sr_device. o smartreflex. o
obj - $(CONFIG_OMAP_SMARTREFLEX_CLASS3)           += smartreflex - class3. o
obj - $(CONFIG_TI816X_SMARTREFLEX)                += smartreflex - ti816x. o

#编译参数
AFLAGS_sleep34xx. o                               := - Wa, - march = armv7 - a
AFLAGS_sleep3517. o                               := - Wa, - march = armv7 - a
AFLAGS_sleep814x. o                               := - Wa, - march = armv7 - a

#用于电源管理的 debug
ifeq ( $(CONFIG_PM_VERBOSE),y)
CFLAGS_pm_bus. o                                  += - DDEBUG
endif

endif

#与芯片相关的 PRCM 数据和代码
# PRCM
obj - $(CONFIG_ARCH_OMAP3)                        += prcm. o cm2xxx_3xxx. o prm2xxx_3xxx. o
obj - $(CONFIG_ARCH_TI81XX)                       += prcm. o cm2xxx_3xxx. o prm2xxx_3xxx. o \
                                                   cm81xx. o
```

```
#电源域的管理框架
# OMAP powerdomain framework
powerdomain - common                      += powerdomain. o powerdomain - common. o
#与芯片相关的电源域代码,主要是芯片相关的数据和特殊操作接口
obj - $(CONFIG_ARCH_OMAP3)                += $(powerdomain - common) \
                                          powerdomain2xxx_3xxx. o \
                                          powerdomains3xxx_data. o \
                                          powerdomains2xxx_3xxx_data. o
obj - $(CONFIG_ARCH_TI81XX)               += $(powerdomain - common) \
                                          powerdomain2xxx_3xxx. o \
                                          powerdomains3xxx_data. o \
                                          powerdomains2xxx_3xxx_data. o
#时钟域管理框架
# PRCM clockdomain control
obj - $(CONFIG_ARCH_OMAP3)                += clockdomain. o \
                                          clockdomains2xxx_3xxx_data. o
obj - $(CONFIG_ARCH_TI81XX)               += clockdomain. o \
                                          clockdomains2xxx_3xxx_data. o
#时钟框架的代码,主要是各种时钟的操作实现
# Clock framework
obj - $(CONFIG_ARCH_OMAP3)                += $(clock - common) clock3xxx. o \
                                          clock34xx. o clkt34xx_dpll3m2. o \
                                          clock3517. o clock36xx. o \
                                          dpll3xxx. o clock3xxx_data. o
obj - $(CONFIG_ARCH_TI81XX)               += $(clock - common) clock816x_data. o \
                                          clock814x_data. o clock81xx. o ti81xx_vpss. o \
                                          fapll_ti816x. o adpll_ti814x. o

#芯片内各个接口模块的具体数据,主要对应资源的信息
# hwmod data
obj - $(CONFIG_ARCH_OMAP3)                += omap_hwmod_3xxx_data. o
obj - $(CONFIG_ARCH_TI81XX)               += omap_hwmod_81xx_data. o

#某些芯片包含为外部 IO 模块提供的 MMU(比如 DM 3730 中的 Camera 接口),相关的操作接口
#实现在这里
obj - $(CONFIG_OMAP_IOMMU)                += iommu2. o

iommu - $(CONFIG_OMAP_IOMMU)              := omap - iommu. o
obj - y                                   += $(iommu - m) $(iommu - y)
#I2C 控制器的底层操作接口
i2c - omap - $(CONFIG_I2C_OMAP)           := i2c. o
obj - y                                   += $(i2c - omap - m) $(i2c - omap - y)
```

```
#OMAP3 系列芯片和 DSP 通信的接口
ifneq ( $(CONFIG_TIDSPBRIDGE),)
obj-y                                          += dsp.o
endif
#PCIe 底层操作接口
obj-$(CONFIG_PCI)                              += pcie-ti81xx.o

ifeq ( $(CONFIG_PCI_DEBUG),y)
CFLAGS_pcie-ti81xx.o += -DDEBUG
endif

#板级特殊代码,只保留 DM 3730 开发板级和 DM 81XX 系列的文件
# Specific board support
#board-generic 此文件是通过 Device Tree 来定义板级特殊数据,后续文件都是文件中明确定义
#板级数据
obj-$(CONFIG_MACH_OMAP_GENERIC)                += board-generic.o
obj-$(CONFIG_MACH_OMAP3EVM)                    += board-omap3evm.o \
                                               hsmmc.o \
                                               board-flash.o
obj-$(CONFIG_MACH_TI8148EVM)                   += board-ti8148evm.o \
                                               hsmmc.o \
                                               board-flash.o
obj-$(CONFIG_MACH_DM385EVM)                    += board-dm385evm.o \
                                               hsmmc.o \
                                               board-flash.o
obj-$(CONFIG_MACH_TI811XEVM)                   += board-ti811xevm.o \
                                               hsmmc.o \
                                               board-flash.o
obj-$(CONFIG_MACH_TI8168EVM)                   += board-ti8168evm.o \
                                               hsmmc.o \
                                               board-flash.o
#具体接口模块设备,根据自行设计板子中实际使用的设备进行配置
usbfs-$(CONFIG_ARCH_OMAP_OTG)                  := usb-fs.o
obj-y                                          += $(usbfs-m) $(usbfs-y)
obj-y                                          += usb-musb.o
obj-y                                          += usb-ehci.o

onenand-$(CONFIG_MTD_ONENAND_OMAP2)            := gpmc-onenand.o
obj-y                                          += $(onenand-m) $(onenand-y)

nand-$(CONFIG_MTD_NAND_OMAP2)                  := gpmc-nand.o
obj-y                                          += $(nand-m) $(nand-y)
```

```
#具体的以太网设备接口,主要是 OMAP 系列处理器使用,由于 OMAP 系列处理器中不包含以太
#网控制器,需要通过同步硬件接口连接外部的以太网控制器实现以太网功能。以下 SMC91X
#和 SMSC911X 是 OMAP 开发板上使用过的以太网控制器
smc91x - $(CONFIG_SMC91X)                  := gpmc - smc91x. o
obj - y                                    += $(smc91x - m) $(smc91x - y)
smsc911x - $(CONFIG_SMSC911X)              := gpmc - smsc911x. o
obj - y                                    += $(smsc911x - m) $(smsc911x - y)

ifeq ( $(CONFIG_MACH_OMAP3EVM),y)
evm - camera - $(CONFIG_VIDEO_OMAP3)       := board - omap3evm - camera. o
obj - y                                    += $(evm - camera - m) $(evm - camera - y)
endif
```

了解 mach - omap2 之后就该来认识 plat - omap 目录了。该目录主要为各种设备驱动提供统一的接口操作，以此屏蔽芯片的特殊操作，比如 GPIO、DMA、时钟等。早期的 Linux 内核中并没有提供这些设备的统一操作接口，随着 Linux 内核的发展，逐渐增加了这些设备的统一接口，该目录存在的意义就必然减小。

至此可以明确从芯片的复杂度来看，DM 3730 是最复杂的，相对于其他芯片来说其技术含量也是最高的。可以说如果能深刻理解 DM 3730 及其 Linux 内核代码，那么除了个别 DM 3730 没有包含的设备接口驱动外，再看其他芯片的代码都是游刃有余的。因此笔者会以 TI 官方发布的 Android 开发包中 DM 3730 的内核代码为主，进行 Linux 内核及设备驱动的剖析，但是为了避免局限性，必要时会扩展到其他芯片，以及新版本的内核进行说明。

2.5 小结

本章着重介绍了 TI 的各种嵌入式处理器及其特点。并从电源管理出发深入分析了电源管理技术以及各个系列芯片的实现细节，从中可以体会到芯片设计本身是软件硬件、性价比、能耗比等因素综合考虑的结果。

另外本章对芯片相关的内核代码的结构进行了分析。从结构分析中可以看出，硬件实现的概念或模块在软件中基本都有对应的实体。软件实现中为了降低耦合度减少冗余代码会进行进一步的抽象，形成统一的接口和框架。从这点出发就能更好地理解处理器的内核代码。

第 3 章　Linux 内核框架探究

谈到 Linux，首先想到的是类似 Ubuntu 这样复杂的系统。说到 Linux 系统复杂，确实如此，光是受欢迎的桌面发行版本就超过十种之多。要是算上各种各样的嵌入式应用的 Linux 系统，估计要超过百种之多。面对如此复杂的系统，没有必要害怕，数量上虽然多，但是整个系统的框架却是相同的。Linux 系统框架如图 3-1 所示。

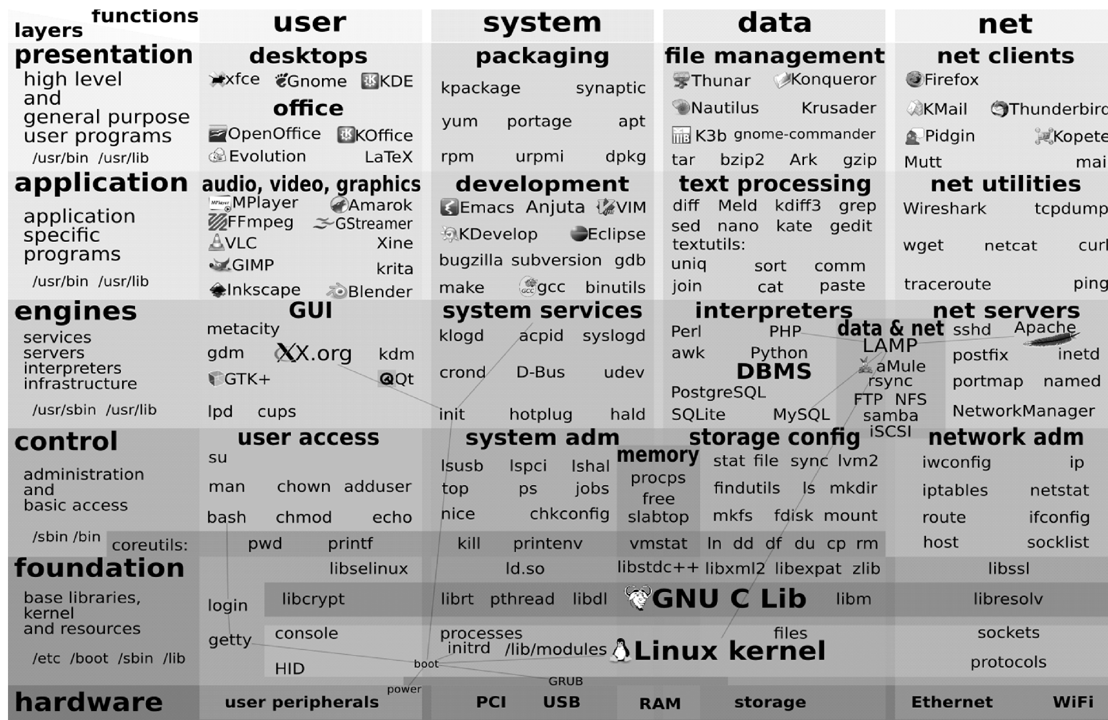


图 3-1　Linux 系统框架

图 3-1 中可以看到与上层的复杂多样相比，基础层（foundation）的核心是简单唯一的 Linux 内核（Linux Kernel 4）。也正是因为 Linux 内核的唯一性，各个不同发布版本拥有相同的框架。Linux 内核是在整个 Linux 系统的最底层，它负责管理硬件，运行用户程序，并保持系统整体的安全性和完整性。虽然 Linux 内核在 Linux 系统中是很小的一部分，但它是 Linux 系统的核心，并决定了整个系统的性能和效率，在整个 Linux 系统中起着独一无二的作用。可以说是 Linux 系统的根和灵魂。

图 3-1 中看到 Linux 内核还有另一层面的含义。从层次的角度考虑，底层的模块都是提供服务的，服务要能够满足应用的各种需求。应用需求的变化必然对底层提出更多的要求，因此底层的服务不能固定不变，也是要发展的。这对于 Linux 内核同样适用。当然 Linux 内核毕竟是在硬件之上的层次，所以应用需求的变化发展不能完全依赖于 Linux 内核解决，一

部分的实现还是要硬件支持才行（比如虚拟化技术）。

可见整个系统的发展是一个综合的结果，对其中一部分的研究也应该综合考虑。所以对 Linux 内核的剖析、学习和研究也不能独立于系统进行，而是要综合考虑应用、内核和硬件等各方面的信息和内容。这样才能更全面、深刻地理解 Linux 内核。

3.1 内核框架概述

关于 Linux 系统框架在图 3-1 中已经比较详尽的展示，如果将其中的层次关系进行简化就得到图 3-2。

从图 3-2 中可以理解 Linux 内核就是将硬件的功能抽象出来，为用户的应用程序提供各种系统服务。这些系统服务在内核中必然会有体现，系统服务本身就是应用层需要的各种各样的功能（function）。关于功能，图 3-1 中也有一些体现，如用户管理、系统管理、数据管理和各种网络功能。这几部分完全是从用户使用的角度考虑的，而 Linux 内核的一个重要功能就是管理硬件，在其功能中必然要体现硬件的各个功能。所不同的是 Linux 内核不是将硬件的各个功能一成不变的体现给应用层，而是需要遵循应用层的功能需求进行逻辑的转换。从这个角度理解 Linux 内核，其主要功能就是管理硬件、将硬件的资源进行合理的抽象并开放给应用层，应用层则无权直接访问硬件，必须通过操作系统来完成相应的功能。这种权限划分和转换是以处理器权限分级为基础来实现的（应用层处于普通权限，而操作系统则有特权）。当然 Linux 内核是需要支持多用户的，考虑用户的因素，就需要在各种物理资源和抽象资源中加上与用户相关的属性（比如资源的拥有者 owner，权限等）。与用户相关的属性是与 Linux 内核中整体安全性相关的，这部分功能通常和硬件具体功能关系不大，而是附加在逻辑功能之上的属性。本书是以嵌入式系统为基础进行说明，所以将重点放在硬件设备的具体功能以及 Linux 内核在硬件之上的具体实现方面。安全相关的逻辑属性及功能不会进行说明。

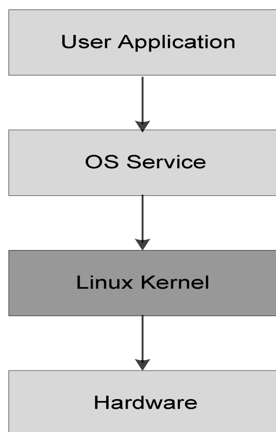


图 3-2 Linux 系统简化层次图

Linux 内核的整体框架如图 3-3 所示。图 3-3 清晰地展现了 Linux 内核的实现层次，以及应用层功能和硬件功能的对应关系。通过该图，可以了解 Linux 内核是如何通过各个层次的抽象，将硬件功能转换成应用层实际需要的功能。

3.1.1 Linux 内核的层次分析

通过分析图 3-3，可以了解 Linux 内核是如何一层一层的将设备功能转换为用户功能的。按照这种层次的角度可以将 Linux 内核分为五层，分别如下：

① user space interfacea：该层是 Linux 内核直接面向用户层的接口，实现用户层需要的各种功能。该层在各个 Linux 内核版本中尽量保持一致。该层可以屏蔽各个版本内核在功能实现的细节差别，从而为应用层提供统一接口，降低应用层和内核之间的耦合度。

② virtual subsystems：该层为虚拟子系统层。所谓虚拟实际是一种高级抽象，是将下层

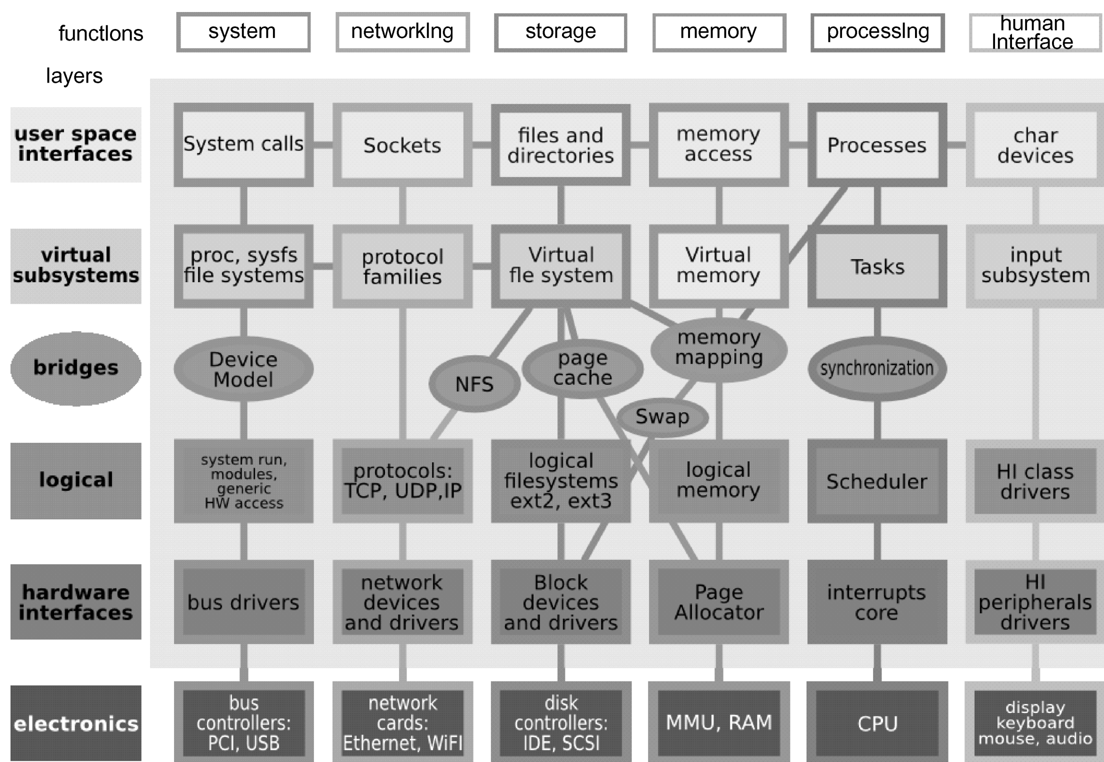


图 3-3　Linux 内核整体框架

（逻辑层）不同的具体实现中体现的数据和操作再做进一步抽象，单独形成的一层。该层主要目的就是为了形成实现无关的一层，使得整个系统不依赖于具体的逻辑实现。使上层系统从复杂多变的实现中解放出来。不仅提高了系统实现的灵活度，也降低系统耦合度。

③ bridges：桥梁层。Linux 内核中各个功能模块之间不是独立的，会有交叉功能的需要，比如文件需要映射到虚拟地址空间，网络文件系统的实现，这些功能都是需要跨越不同功能的实体。当跨越不同功能实体时，需要在二者之间建立关联，也就是桥梁。另外某些功能本身就需要有很多的实体，比如在处理器之上的执行实体进程（process），而功能内部的多个实体之间也需要相互沟通，这样就需要桥梁层的实现来解决这个问题。总之当需要沟通或者转换时就需要在 bridges 层实现。

④ logical：该层负责功能的逻辑实现。某一个具体的功能可以有很多种实现，而这些具体的实现都是在 logical 层。比如 Linux 的文件系统有 ext2 和 ext3 不同的实现逻辑，而 ext2 和 ext3 都是在 logical 层中。可以说该层是 Linux 内核中最多样的一层，但是这些多样实现通过设计良好的接口去封装，使功能之间并没有耦合，因此没有增加系统的复杂度。

⑤ hardware interfaces：该层是 Linux 内核的最底层，其直接面向具体的硬件设备，是将硬件的具体功能提供给 Linux 内核的接口层。该接口层将设备的功能分类，并抽象出统一接口给 Linux 内核使用。Linux 内核通过统一的接口操作设备，这样就屏蔽了各个设备的差异，从而降低了和硬件的耦合关系。

Linux 内核各层体现不同程度的抽象，越往上层抽象程度越高，覆盖面越大，也越接近于应用的需求；越往底层其体现的特殊性就越多，覆盖面越小，越接近于具体的设备。有了

这些层次的概念，对了解 Linux 内核是很有帮助的，在具体的实现中需要明确模块是在哪一层，具体管理的抽象实体是什么，掌握了这些，基本的框架就能理解了。

现在可以通过具体功能的例子，看看 Linux 内核是如何通过各层的抽象，使用设备实现用户功能的。

1. 程序执行（processing）功能

processing 功能对应的硬件实体是 CPU。CPU 是执行指令的单元，它可以接收外部事件（通过中断 Interrupt）。指令执行需要操作数，相同操作指令结果的多样性通过操作数的变化来实现。操作数可以从 memory 中读取，这就可以通过使用相同的指令而从不同的 memory 地址取操作数，实现操作结果的变化，从而增加了系统的灵活性。另外执行逻辑的变化可以通过跳转指令完成，同样通过从 memory 读取指令地址，可以实现多变的执行逻辑。这样不同的操作数和跳转地址组合成一块 memory，这些不同的 memory（称作栈）就可以产生多变的系统执行流程和结果。CPU 通过栈指针寄存器可以指向不同的 memory 的位置。为了减少寄存器，软件对该寄存器操作流程进行特殊规定，这样通过硬件和软件结合，使得 CPU 可以在不同的指令执行流程和操作数之间切换。从 CPU 的角度来看，它就是按照指令的要求，在不同的执行流程中进行切换并执行指令；从操作系统的角度考虑，这些不同的软件执行流程就是线程（thread）。Linux 内核为了简化实现，将共享资源的进程作为线程。图 3-4 展示了 Linux 内核中进程间（不同的执行流程）的切换，就是通过内核栈的改变来实现的，具体是通过 switch_to 函数实现。

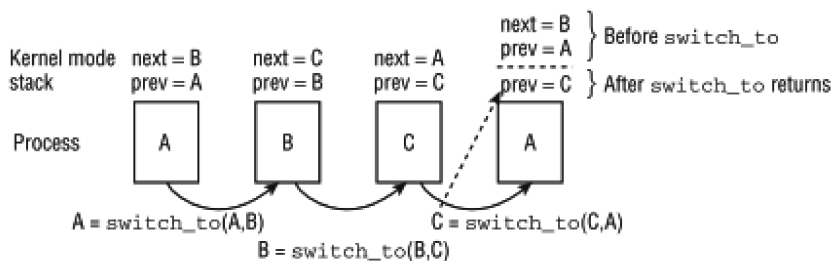


图 3-4　Linux 进程切换实现逻辑

有些执行流程的切换是由 CPU 自动完成的，比如中断、异常，这些都是系统为了满足外部事件，以及安全性、容错性的需求。有了中断之后系统就能对外部事件进行管理。由于不同的 CPU 中断处理存在差异，这就需要软件上抽象出中断管理的接口。图 3-3 中硬件接口层中的 interrupt 核心就是实现该功能的，具体的 CPU 只要实现相应的接口就可以在 Linux 内核中实现对中断的管理。

考虑到外部事件需要对执行流程进行转换，另外不同的应用执行流程之间需要切换，这就要在 Linux 内核的逻辑层抽象出调度器这一逻辑功能。该功能负责在不同线程之间选择合适的线程来让 CPU 执行（即切换到相应的执行流程中），以完成对应的任务。注意调度器实际是个选择器，其按照一定的算法选择合适的任务，具体的切换执行在 Linux 内核中是由 switch_to 函数实现的。Linux 作为多用户系统，调度器的算法和性能十分重要，因为其影响到所有用户的感受。

应用层需要 Linux 内核能够执行应用程序，而在应用程序中不仅包含指令执行流程还包含程序操作的资源，比如打开的文件，地址空间中的内容以及对文件的映射等。这些需要

Linux 内核将它们作为一个整体进行抽象，这就形成了进程的概念，这在 Linux 内核中就是虚拟子系统层中的 task。同样在应用接口层也需要相应的接口对 task 进行操作，比如创建、设置不同的执行指令等。

大型的任务需要分多个线程或者进程来完成，这样可以降低系统复杂度，这就提出了多个线程或进程之间沟通并同步的需求，这部分就是 bridges 层的功能。

2. 内存管理（memory）功能

内存管理主要是对 RAM 的管理，是 Linux 内核中很大的一部分功能。应用层的程序需要任何时刻都能以应用程序自己设定的地址访问数据，并且应用程序应该只能知道自己的地址。这样从应用程序自身地址的角度考虑，不同的应用程序必然会使用相同的地址做不同的事情。而物理内存作为唯一资源就无法实现同一地址存放不同的内容。这就产生了矛盾，也就需要应用程序的地址和物理内存的地址不能是相同的概念，这之间的鸿沟需要通过一种转换进行解决，这样就产生了 MMU。由于应用程序的地址空间不直接对应于物理内存的地址，所以在 Linux 内核中形成虚拟地址（virtual address）。Linux 内核需要对所有应用层任务的虚拟地址空间进行管理，这部分就形成了 virtual 子系统层中的 virtual memory。当然对虚拟地址空间的管理不只是管理地址，同时还要管理相应空间中存放数据及数据的属性，不同性质的数据在地址空间中不应该有交叉和重叠。Linux 虚拟地址组织结构如图 3-5 所示。注意虚拟地址空间是和 task 相关联的，并且其中的数据是有分别的，每块虚拟空间中存放的数据都有不同的属性和操作方式。

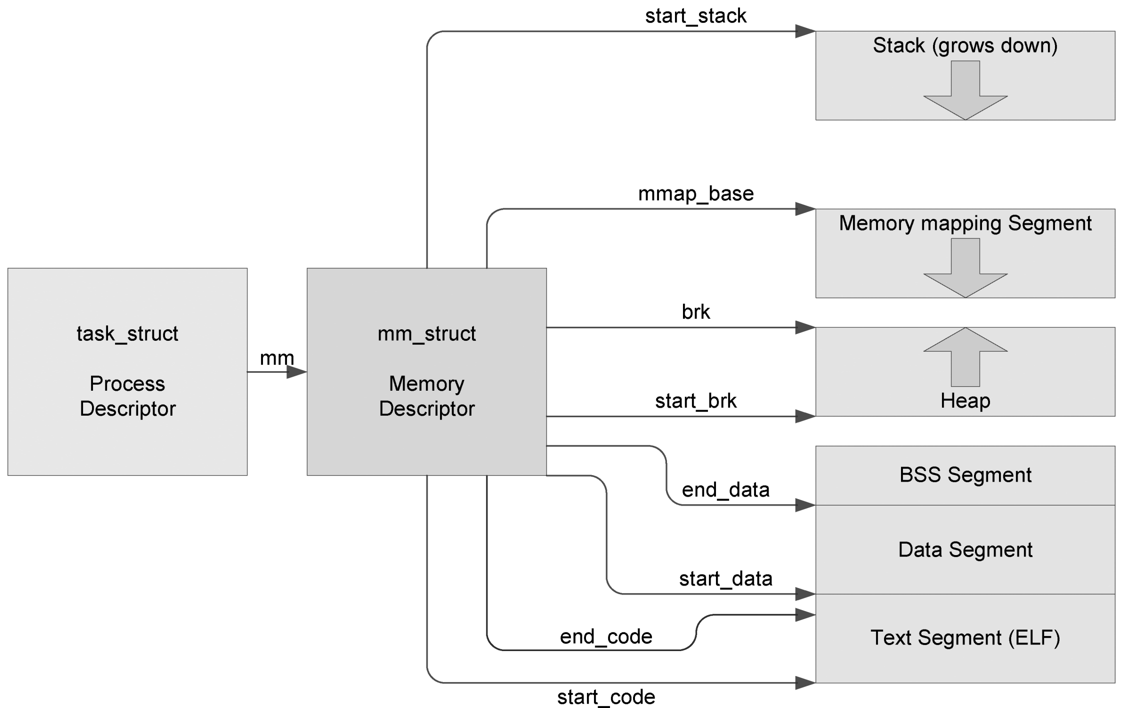


图 3-5　Linux 虚拟地址组织结构

物理内存是一个被动器件，只是接受地址和数据进行读写的操作，具体数据的属性并不

关心。物理内存究竟如何管理是个需要仔细考虑的问题。管理总是要考虑粒度的，如果过大就会有粒度内部浪费的问题，如果过小就会消耗更多的资源来进行管理。权衡下来，硬件的体系结构通常将最小的内存管理单元定义成页（page），把页大小（即粒度）的设置留给系统软件完成。可以由系统软件根据管理的物理地址空间以及内存的大小来进行设置。ARM就可以设置 4 KB 或者 64 KB 不同的页大小。

当然仅是页管理不能满足系统所有模块对内存使用的需求，这就需要在页的管理之上将数据进行结构的和细粒度的管理，这在 Linux 内核逻辑层中形成了 slab 内存管理，为内核其他模块提供管理内存的结构。slab 分为 kmem_cache 和 kmalloc 两种形式，kmem_cache 提供结构化的内存管理，kmalloc 则提供细粒度的内存管理。

虚拟地址到物理地址的变换，硬件上由 MMU 来实现，其中也会涉及管理页的大小，ARM 下 MMU 的实现逻辑如图 3-6 所示。其中各级页表也都是存放在内存中，一级页表的地址存放在系统寄存器中，对进程的切换时同样需要切换页表。

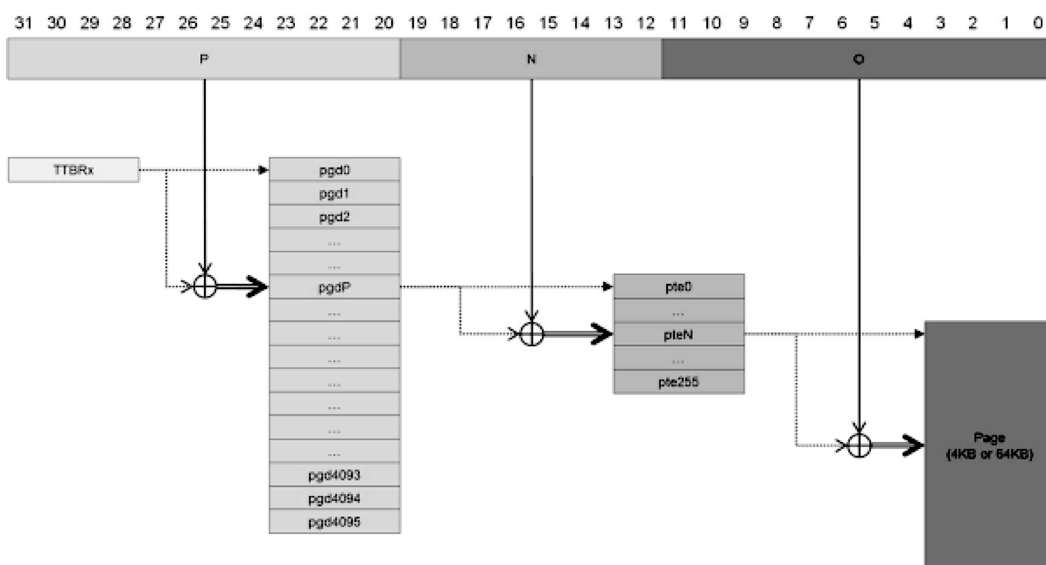


图 3-6　ARM MMU 实现逻辑

需要注意的是对 CPU 来说，只要有页表 MMU，就能通过虚拟地址完成实际物理地址的具体操作，CPU 本身并不需要对虚拟空间进行管理。虚拟空间管理则是完全针对虚拟地址进行的。由于虚拟地址和 task 是关联的，所以实际上，虚拟空间管理在 Linux 内核中就是对所有用户进程虚拟空间的管理。

3. 存储（storage）管理

存储管理针对的硬件设备是外部磁盘，如 IDE、SATA、SCSI 等。随着嵌入式的发展，外部存储设备又增加了 Nand、SD 卡等。这些存储设备的特点是大多硬件要求以块为单位进

行操作，这就已经定下了管理粒度即块。存储设备的这个特性自然的形成了 block device 和 block driver 作为硬件接口层。该硬件接口层只是完成以块为单元的数据操作。

数据在存储设备中如何组织分配等更细致的问题，就需要逻辑层处理。这部分处理细化出各种文件系统，当然某些数据库也是在逻辑层实现的。文件系统屏蔽了块设备的细节，而且将数据组织并抽象出文件的概念。为了更好地组织、管理和查找文件，可以将多个文件放入一个目录，这又形成了目录的概念。文件系统管理的是这些逻辑层面的概念，以及对应到实际设备块的映射逻辑。具体的块操作完全由设备接口层完成。

由于逻辑层有各种各样的文件系统，则需要在虚拟层再进行抽象，屏蔽各种具体实现的差异，从而形成虚拟文件系统模块。这样用户接口层就可以通过虚拟文件系统的统一接口对不同的文件系统进行操作。

又由于外部磁盘读写比较慢，而内存读写速度要快得多，这样在逻辑上做成页缓冲可以提高性能。在层次上页缓冲涉及虚拟文件系统和页管理，属于交叉功能，所以应该在 bridges 层实现。

4. 网络（networking）管理

网络管理最底层是网络设备，相当于网络协议分层的物理层。其他层的网络协议是在物理层之上的，同样的网络设备上可以传输不同的上层网络协议封装的数据。所以在层次上，物理层之上的协议归入逻辑层，而物理层的设备归入设备接口层。

再向上同样需要抽象出对协议族的管理和对应用层的 socket 接口来简化用户的操作。

5. 系统（system）管理

Linux 内核中有各种各样的系统。为了提升系统在不同情况下的适应能力，通常各种系统都会有一些参数和状态信息提供给用户或者系统管理人员，进行查看、修改、调优。这就需要 Linux 内核设计统一的接口，可以让各个系统模块的开发者来添加相应的参数，同样需要为应用开放统一的操作接口和方式。随着 Linux 的发展，最终产生了 proc 和 sysfs 两个大的接口系统进行该类操作。该部分实现会散布在各个系统功能中，本身的层次概念可以参考图 3-3，但不是必需的。

6. 用户界面（user interface）管理

用户界面直接关系到用户体验的部分，设备也是最复杂多样的。比如键盘、鼠标、显示器、音频设备、摄像头等各种各样的设备，这些设备没有统一的数据格式和标准，如果一定要找这些设备的共同点就是数据有时效性。数据的意义和时间是关联的，任何数据会因为时间的改变而意义不同，用《信号与系统》中的名词就是时变系统。如何在系统中抽象并管理这些设备是比较复杂的问题。Linux 内核大牛们用了一个很精巧的办法，就是首先使用高级别的抽象概念设备文件，该类设备都归结为字符设备（char device）；其次将操作也抽象处理成为统一接口，而将具体设备的属性完全留给底层的设备驱动来进行管理。这样应用层程序就可以通过统一的字符设备（char device）文件进行操作。

下层的实现可以针对设备的特点加入更多的细节进行管理，这样 Linux 内核中就针对不同设备的特点进行归类，形成 input、audio、frame buffer、video 等各种各样的设备类型。这些设备类型的具体实现一般是归入逻辑层。

逻辑层之下就是具体设备的驱动，归入设备接口层。在逻辑层之上，可以将与设备传输的数据进行抽象或者格式化，形成系统统一的规范，比如 input event 处理等，将这部分作为

虚拟层。

这样各个层次紧密结合，既可以适应广泛的设备，又可以对应用提供统一操作接口。当然对具体类型设备的操作要遵循相应类型设备的操作规范。

注意在图 3-3 中只有 char device 出现在用户空间接口层，这并不意味着块设备不可以直接对用户应用程序开放。只是这种开放通常只针对像 fdisk 这种格式化或者文件系统相关的工具。逻辑上块设备更应该是在文件系统之下，作为数据存储的基础。

对各个层进行深刻理解，有助于内核以及驱动开发人员更加明确地理解系统各个功能的实现机制和设计思路，通过理解每层的抽象概念可以更好地跳出细节理解系统。

3.1.2 Linux 内核模块间关联

对于复杂的软件系统，在框架设计时都会有层次的概念。系统框架的层次设计有很多好处，如层次化设计使得系统层次分明，可以提高重用性，使得系统耦合度低。Linux 采用层次化的设计自然也有这些优点。但是，软件系统设计，不是简单的纵向层级就能完全解决所有的问题，软件系统中通常都会有横跨多个纵向模块的功能（称作横切功能），比如日志等。Linux 内核中也不乏这样的功能，比如 sys 文件系统等。另外 Linux 框架也是不断发展的，为了满足各种性能的需求，不能为了层次而层次，而是需要将整个系统打薄，可以看到 Linux 内核很多时候会有模块进行跨多层的联系。Linux 内核模块之间详细的关联如图 3-7 所示。

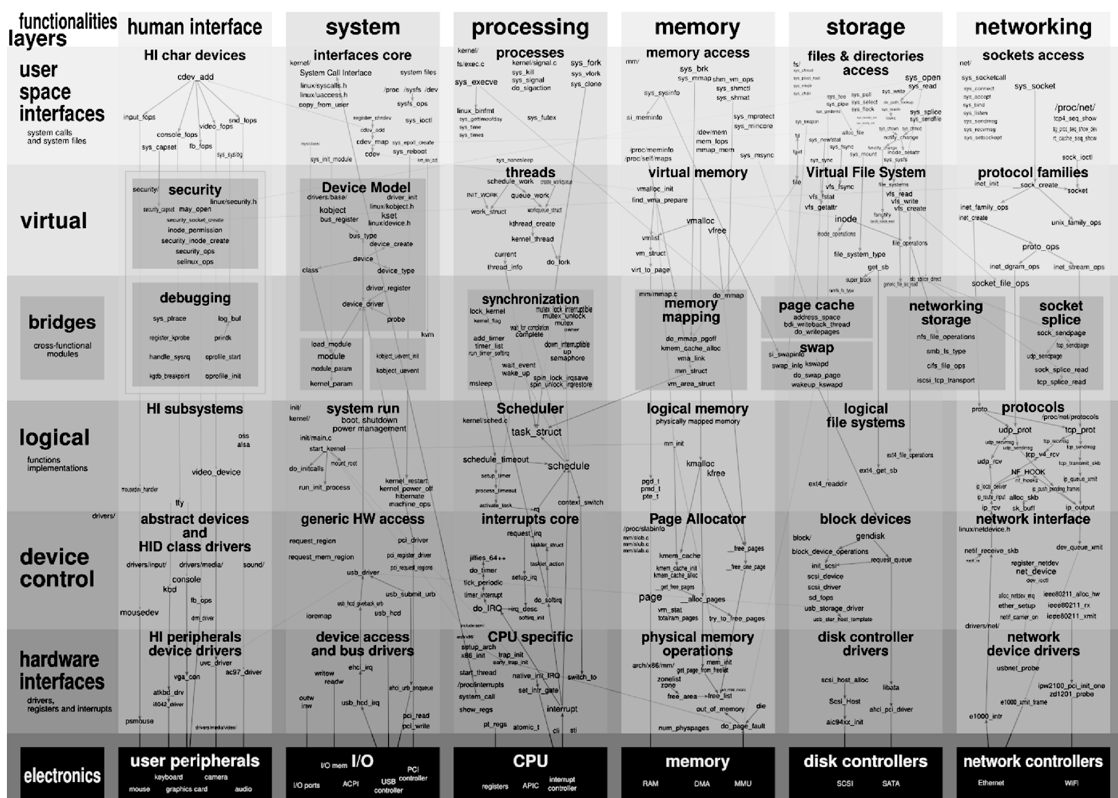


图 3-7 Linux 内核模块间关联图

图 3-7 引自 http://www.makelinux.net/kernel_map/，读者可以直接下载了解更详细的内容。看到 Linux 内核复杂的关联，可能有些读者会有恐惧感。但是不要担心，理解这些关联的基础是 Linux 内核的层次，这在前面已经做了详细介绍，后续的章节还会梳理这些关联的关系，这里先有个概念。

3.2 需求探究

探讨需求首先要考虑的就是需求从哪里来的问题。这就涉及用户是谁的问题。Linux 内核的用户是谁呢？Linux 内核的用户不是 Linux 桌面系统的使用者，主要是两类程序开发人员，一类是应用层的开发人员，另一类是内核及驱动的开发人员。

这两类开发人员看 Linux 内核系统的视角是不同的，应用层开发人员是从外部看 Linux 内核，更多的是将内核作为黑盒；而内核及驱动开发人员是从内部看 Linux 内核，需要了解 Linux 内核的内部实现机制。

视角不同自然需求就不同。应用开发人员需要 Linux 内核提供的功能，主要在用户管理（包括用户界面和用户账户管理）、系统管理（涉及资源分配和时间管理等）、数据管理（通常通过文件系统相关接口实现）和各种网络功能几个方面。另外 Linux 内核设计之初就考虑兼容 UNIX 的应用，所以要求与 UNIX 功能接口一致，不能在不同版本中功能有变化。针对这些特点，Linux 内核提供和 UNIX 相同的应用层接口即系统调用。由于 UNIX 本身并不是图像化系统，所以设计时并没有考虑图形化操作的接口。所以 Linux 内核在开始的时候并没有像 Windows 那样考虑这部分功能，在内核也没有图形渲染的驱动，只有通过字符设备这种抽象的功能和 ioctl 这种扩展性极强的系统调用，来进行各种扩展。随着系统的发展，现如今也提供了 DRM（direct rendering manager）图形渲染等功能，来完善图形化操作。当然这部分功能并不是通过增加新的系统调用，而是通过字符设备和 ioctl 系统调用来共同完成的。可见在设计系统时，关注主要功能并保留一个强大的扩展能力是十分重要的。

应用层开发人员还有对设备操作的需要，但是应用开发人员不想了解设备的具体细节，只是想使用设备完成一定的功能。对这些功能的需求，还要求这些功能的实现在任何硬件上都是相同的，也就是硬件无关或者说屏蔽各种硬件的差异。这部分需求，就要求 Linux 内核中提供给应用层的资源，要是虚拟的逻辑资源，不能是物理资源，通过转换来实现对应用层统一的视角。对于应用层来说硬件的无关性需要操作系统支持各种各样的设备，可以说要支持所有的设备，这样应用层可以只关注其业务的实现，而不关心设备寄存器的读写逻辑。

应用层还有对安全性的需求，不同的用户所拥有的资源是有限的，并且不能超越权限访问其他用户的资源。

Linux 内核有一类特殊的用户，就是系统管理员，这些用户需要对内核有一定的了解，并对内核进行相应的设置。这就需要内核很多模块开放设置，设置分为启动参数设置和运行时参数设置。设置很大程度上是进行系统调优，是为了性能。Linux 内核也要满足性能需求，需要更快、更高、更强。

内核和驱动开发人员需要 Linux 内核提供基本功能和框架的支持。他们更多是需要 Linux 内核有良好的框架设计和性能，可以使得他们开发的模块容易集成到内核系统中，并且彼此隔离对其他模块没有影响。从内核和驱动开发人员的角度，功能可以分为核心功能和

设备相关功能两个方面。

3.2.1 对内核核心的需求探究

谈到内核核心需求，先回忆一下最小系统。最小系统包括主处理器（可以理解为CPU）、内存（RAM）、外部存储设备（最小系统中是NAND Flash）、电源管理芯片以及时钟晶振。内核核心功能应该能满足最小系统的需求。从最小系统的硬件出发，考虑内核应该满足的功能，包括程序管理功能、内存管理功能和存储管理功能，当然也对应到这些功能的各个层次的实现（在Linux内核层次分析中做过讨论）。另外针对时钟晶振的硬件，内核还应该提供时间管理和定时器管理相应的功能。

最小系统中最核心的硬件是CPU和RAM，在系统执行期间这两个硬件是必需的。所以这两部分的管理可以说是Linux内核的重中之重，对性能等各方面的影响也是最大的。对这两部分的性能需求非常苛刻，这也就是这两部分在Linux内核中，直到今天还在不断改进的原因（相关改进如将中断处理分成前半部和后半部，抢占式调度，调度算法演变为O(1)算法，采用CFS调度器或Deadline调度器等新式调度器，内存管理支持大容量内存，大内存页等等）。可以预见未来这两部分功能仍然是改进的重点，但是改进的方向应该在SMP和多核并行运算，以及锁的效率上。效率至上在这两部分中体现特别明显，只要性能有提升，哪怕是面目全非的完全重写，也是必须做的。图3-8中展示了Linux内核这两部分可供各个模块使用的功能，从中会看到很多熟悉的函数，后面会进行比较全面的介绍。

从图3-8中可见，体系结构无关的代码主要是在kernel以及mm两个目录中，而与体系结构相关的代码则在arch/xxx/和arch/xxx/mm下。可见Linux内核在目录编排上也是有比较多的讲究的，这在后续讨论设备驱动时会有更多介绍。

在图3-8中的memory硬件中除了RAM还有DMA和MMU。通常MMU和RAM一起实现内存管理，而DMA作为底层的一个横切功能，可以作为基础库提供给不同的设备驱动使用。由于DMA在硬件中的广泛使用，内核同样需要在核心部分提供该功能。另外和硬件相关的底层横切功能还包括clock、GPIO和引脚管理等功能。随着Linux的内核发展这些底层的横切功能已经逐渐脱离具体的体系结构形成抽象库，以供Linux内核和设备驱动使用。可见Linux内核核心功能需求也是逐渐发展来满足内核和驱动开发需要的。

3.2.2 对设备管理的需求探究

如果对内核代码进行统计，会发现超过百分之五十的代码都是设备驱动。可见Linux内核支持了大量的设备，也说明设备的多样性支持对于内核的重要性。应用层需要统一的接口对设备进行操作，内核中还需要良好的框架和机制对这些设备进行管理，而很多情况下还需要设备能够被发现并绑定正确的驱动（驱动需要允许动态加载），这些都是需要内核框架的支持。随着嵌入式设备的发展，功耗问题逐渐成为研究的重点，这样在框架方面也需要完整、统一、简单的实现，以对整个设备的使用以及功耗进行管理。所有这些需求都是需要在各种各样的设备上实现，实际就会跨越不同的设备类型，从而形成对各种设备的横切功能。

无论是应用层的设备还是横切功能，在设备管理中都是高层的抽象概念。这些抽象的细节如图3-9所示，图3-9中的设备模型（device model）就是实现横切功能的高级抽象模块，后续会对设备模型进行详细讲解。横切模块通常采用内嵌的方式来具体的实现。kernel对外

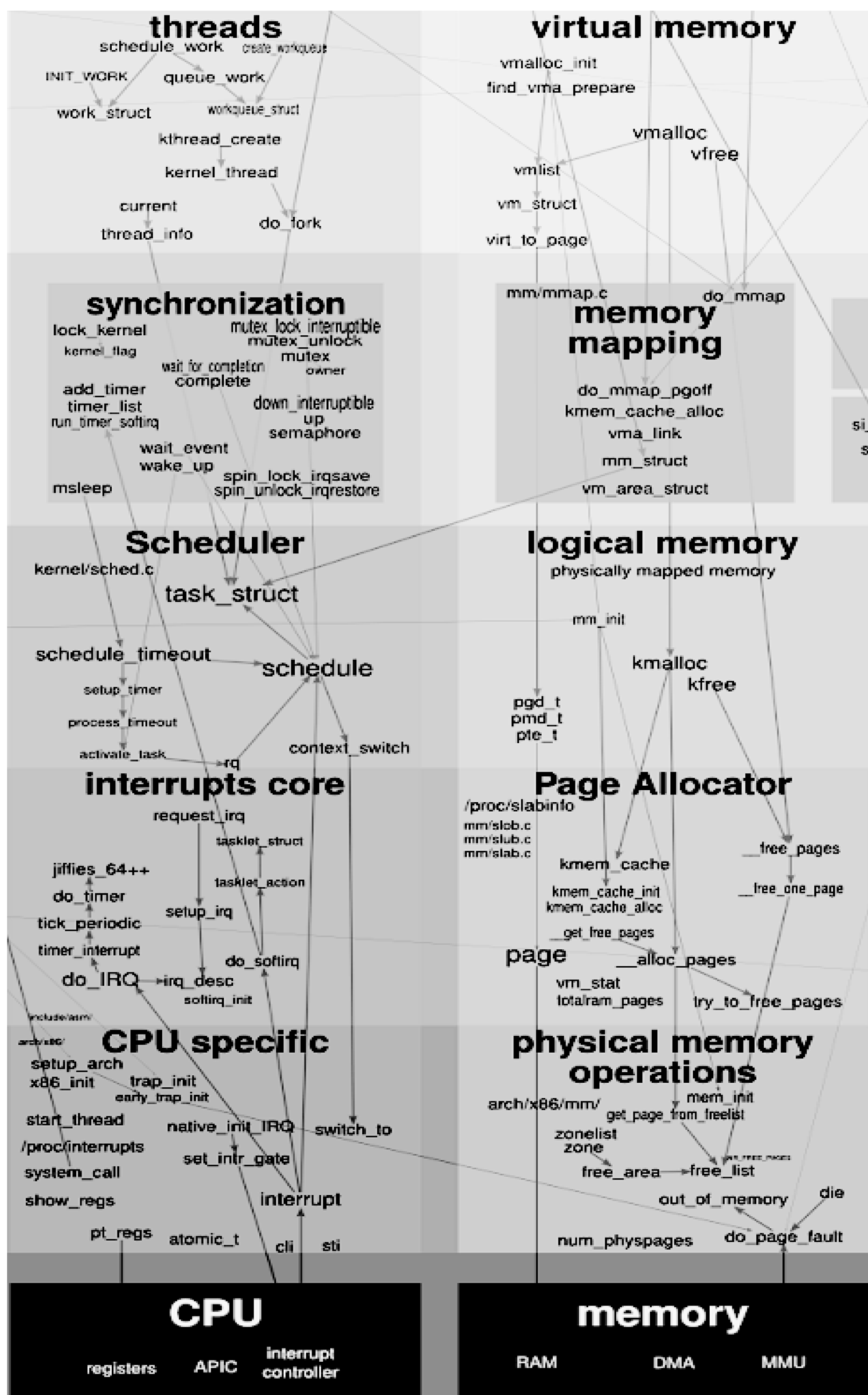


图 3-8　CPU 和 RAM 管理框图

提供了良好的接口 sys 文件系统，可以使用户查看具体的设备信息。关于设备，之前已经介绍用户直接交互的设备主要是人机接口（human interface）设备，人机接口设备主要是通过字符设备框架来做具体的实现。一个良好的设备实现框架，如字符设备框架，也是设备开发的重要需求。其他的需求，如中断框架、地址映射、内存管理、定时器等内核核心功能模块都是支撑驱动开发的基础。还有一些情况某些设备中会有子设备，比如现在的视频设备中会有 I^2C 总线进行控制，这样在视频设备中就需要使用 I^2C 的设备驱动，这就要求各种设备驱动提供良好的接口以便其他设备在需要时使用相应的功能。这些都是对设备及设备开发的需求。

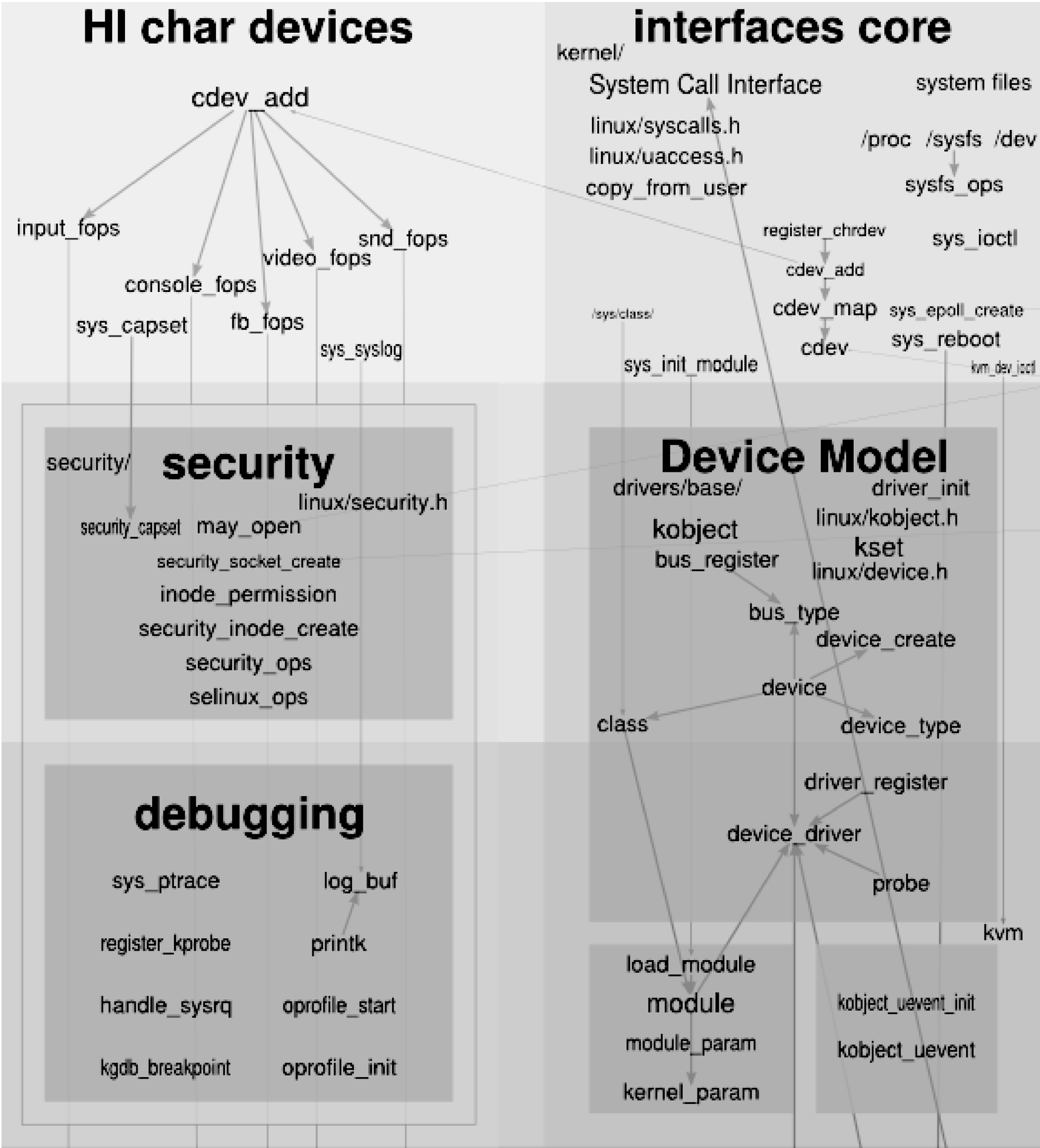


图 3-9　设备模型和抽象设备框图

3.3 按需求的设备分类

通常的分类方式是将设备分为字符设备、块设备和网络设备。这种分类方式在 Linux 内核的层次分析中已经讨论过，三种设备面向完全不同的三种应用，并且它们的数据组织形式以及数据特点还是有着明显的差别的。字符设备的数据可以认为是流式时变的；块设备的数据是以块为数据单元的；网络设备的数据组织形式是和数据链路层的数据组织相关的。这样的分类方式只是在高级抽象层中对主设备相关的分类，对具体的设备究竟如何划分并没有涉及。

谈到设备分类，还会想到 Linux 内核中的 drivers 目录，drivers 目录下的子目录本身就是对设备驱动的分类。图 3-10 是 2.6.37 Linux 内核 drivers 目录下的子目录，目录中可以看到抽象的设备（如 char、block），也可以看到具体的设备（如 media、rtc 等），有些是和设备以及电源管理框架相关（如 base、power、cpuidle、cpufreq、thermal 等），还有基于代码维护考虑创建的目录（如 staging 该目录下的驱动还没有确认合并进代码主干）。看到这么多子目录并不清楚 drivers 目录下的驱动究竟是以什么原则进行子目录的设计，究竟应该如何将设备进行分类。由于 Linux 内核中 drivers 目录下子目录设置有代码管理的原因，每个子目录都对应着 Linux 内核的一个 project。由于 Linux 内核是分布式开发的，通过 porject 进行目录的定义对代码管理来说是比较好的方式。随着 Linux 内核的发展，目录也在不断地增加，有时也会对目录进行整理移动，比如音频相关的框架和驱动就移动到内核的根目录下的 sound



图 3-10　2.6.37 Linux 内核 drivers 目录子目录

目录中。

还有一种设备分类，就是看看内核自己是怎么分的。这就要看看内核开放给我们的接口 sys 文件系统。新的 Linux 内核在提供信息方面已经做得比较完善，相应的设备信息可以通过 sys 文件系统一探究竟。下面是 sys/class 中的信息。

```
dongfeng@ DF - WS /sys/class  $  ls - al
total 0
drwxr - xr - x 52 root root 0 Dec 10 22:05 .
dr - xr - xr - x 13 root root 0 Dec 10 22:05 ..
drwxr - xr - x   2 root root 0 Dec 10 22:05 ata_device
drwxr - xr - x   2 root root 0 Dec 10 22:05 ata_link
drwxr - xr - x   2 root root 0 Dec 10 22:05 ata_port
drwxr - xr - x   2 root root 0 Dec 10 22:05 backlight
drwxr - xr - x   2 root root 0 Dec 10 22:05 bdi
drwxr - xr - x   2 root root 0 Dec 10 22:05 block
drwxr - xr - x   2 root root 0 Dec 10 22:05 bluetooth
drwxr - xr - x   2 root root 0 Dec 10 22:05 bsg
drwxr - xr - x   2 root root 0 Dec 10 22:05 devfreq
drwxr - xr - x   2 root root 0 Dec 10 22:05 dma
drwxr - xr - x   2 root root 0 Dec 10 22:05 dmi
drwxr - xr - x   2 root root 0 Dec 10 22:05 drm
drwxr - xr - x   2 root root 0 Dec 10 22:05 firmware
drwxr - xr - x   2 root root 0 Dec 10 22:05 gpio
drwxr - xr - x   2 root root 0 Dec 10 22:05 graphics
drwxr - xr - x   2 root root 0 Dec 10 22:05 hidraw
drwxr - xr - x   2 root root 0 Dec 10 22:05 hwmon
drwxr - xr - x   2 root root 0 Dec 10 22:05 i2c - adapter
drwxr - xr - x   2 root root 0 Dec 10 22:05 input
drwxr - xr - x   2 root root 0 Dec 10 22:05 leds
drwxr - xr - x   2 root root 0 Dec 10 22:05 mdio_bus
drwxr - xr - x   2 root root 0 Dec 10 22:05 mem
drwxr - xr - x   2 root root 0 Dec 10 22:05 misc
drwxr - xr - x   2 root root 0 Dec 10 22:05 mmc_host
drwxr - xr - x   2 root root 0 Dec 10 22:05 net
drwxr - xr - x   2 root root 0 Dec 10 22:05 pci_bus
drwxr - xr - x   2 root root 0 Dec 10 22:05 power_supply
drwxr - xr - x   2 root root 0 Dec 10 22:05 ppdev
drwxr - xr - x   2 root root 0 Dec 10 22:05 ppp
drwxr - xr - x   2 root root 0 Dec 10 22:05 printer
drwxr - xr - x   2 root root 0 Dec 10 22:05 regulator
drwxr - xr - x   2 root root 0 Dec 10 22:05 rfkill
drwxr - xr - x   2 root root 0 Dec 10 22:05 rtc
drwxr - xr - x   2 root root 0 Dec 10 22:05 scsi_device
drwxr - xr - x   2 root root 0 Dec 10 22:05 scsi_disk
drwxr - xr - x   2 root root 0 Dec 10 22:05 scsi_generic
```

```
drwxr-xr-x  2 root root 0 Dec 10 22:05 scsi_host
drwxr-xr-x  2 root root 0 Dec 10 22:05 sound
drwxr-xr-x  2 root root 0 Dec 10 22:05 spi_host
drwxr-xr-x  2 root root 0 Dec 10 22:05 spi_master
drwxr-xr-x  2 root root 0 Dec 10 22:05 spi_transport
drwxr-xr-x  2 root root 0 Dec 10 22:05 thermal
drwxr-xr-x  2 root root 0 Dec 10 22:05 timed_output
drwxr-xr-x  2 root root 0 Dec 10 22:05 tty
drwxr-xr-x  2 root root 0 Dec 10 22:05 usb
drwxr-xr-x  2 root root 0 Dec 10 22:05 vc
drwxr-xr-x  2 root root 0 Dec 10 22:05 video4linux
drwxr-xr-x  2 root root 0 Dec 10 22:05 vtconsole
drwxr-xr-x  2 root root 0 Dec 10 22:05 watchdog
drwxr-xr-x  2 root root 0 Dec 10 22:05 wmi
```

从中可以发现，sys/class 的信息很多和 drivers 目录下的信息相似，而且信息量更大，细节更多。比如 scsi 就有四个相关的条目，spi 也有三个相关的条目。这是由于总线中会有多种类型设备来保证设备互连。其他部分有些是横切功能（如 dma、gpio 等），有些是用户接口功能（如 sound、graphic 等），总之是包含所有类型的信息。

讨论过数据操作形式的设备分类、驱动目录的分类以及 sys 文件系统分类等方法后，读者对实际设备如何工作、驱动如何实现仍然十分模糊，而且对实际设备的实现层次没有清晰的认识。这里首先来看看 PC 上的硬件层次。PC 上的硬件层次如图 3-11 所示。

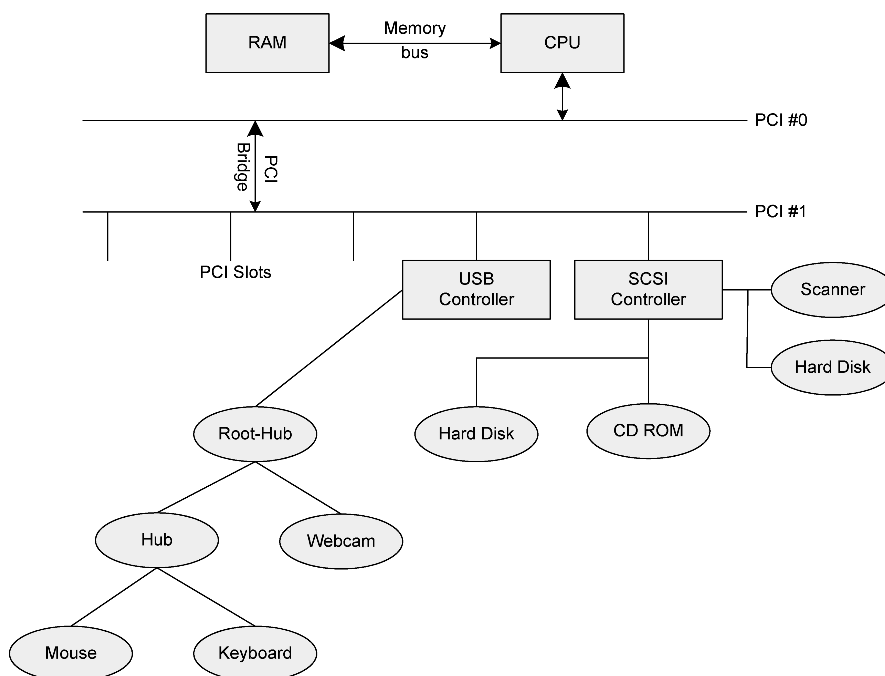


图 3-11　硬件层次图

以图中 Webcam 为例，硬件上从 CPU 到 Webcam 经过了 PCI 总线和 USB 总线，然后才连接 Webcam，这样硬件实际就是 Webcam、USB、PCI 三层。硬件上有了层次的概念，同样的在驱动框架中也应该有对应的层次概念。驱动的层次如图 3-12 所示。图 3-12 中圈起来的部分是实现 Webcam 功能相关的模块，video_device 是 Webcam 视频功能的驱动框架，uvc_driver 则是 Webcam 的具体实现驱动，相应的 usb_driver 就是硬件层中 USB 的具体实现。这里有个问题，为什么没有 PCI 呢？PCI 如何体现呢？驱动软件中已经将 PCI 封装到具体的 USB 总线实现中，毕竟 USB 设备本身并不关心其所在的 PCI 总线的实现细节，通过 USB 的实现屏蔽 PCI 的细节，可以让开发者只关注直接关联的接口实现，降低实现的复杂度。对硬件的层次关系与驱动的层次关系进行比较可以发现，硬件的层次关系是站在从设备互联的角度从处理器到功能设备，而软件驱动的层次角度是从功能到设备互联，刚好是反向过程。这样从硬件的层次需求出发来理解设备驱动会更清晰、更明确。

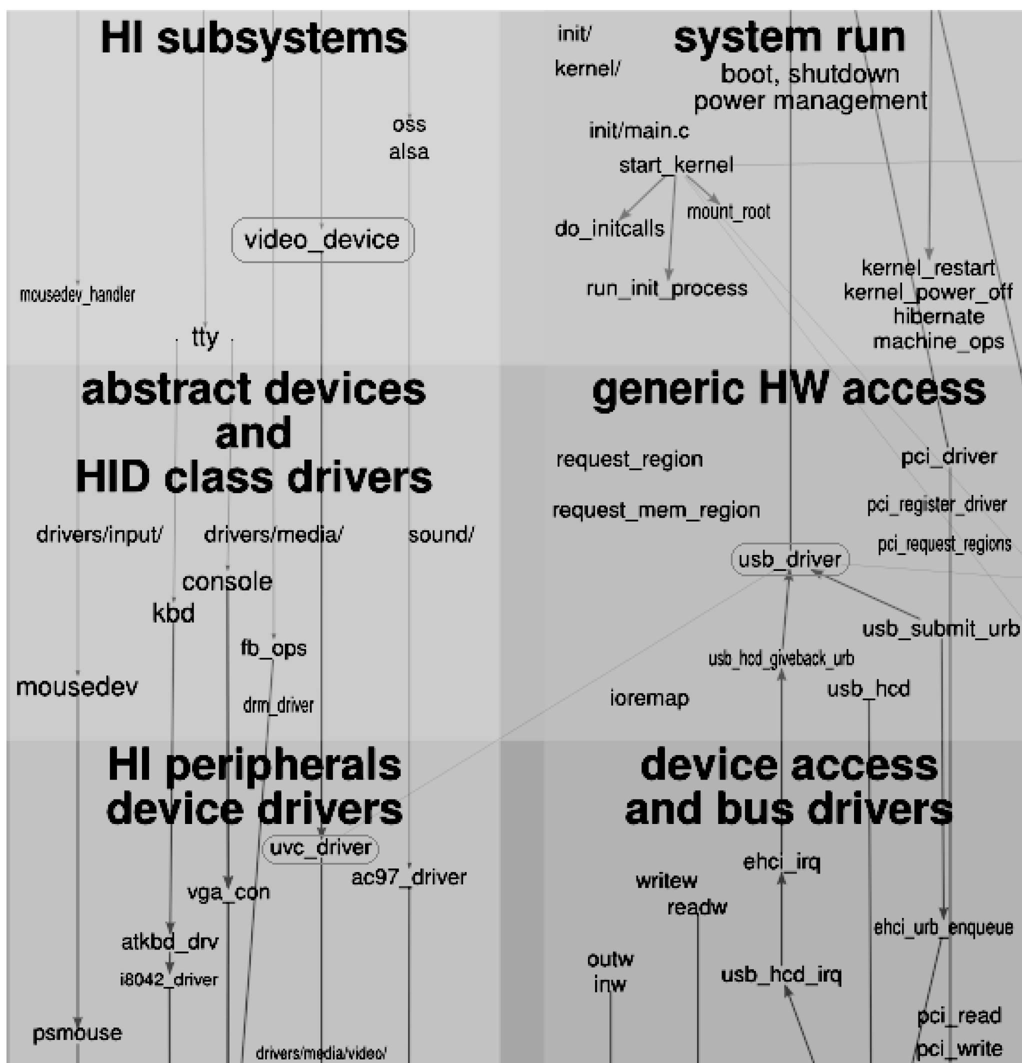


图 3-12　驱动层次

看过 Webcam 的例子后，再重新考虑设备分类的问题，硬件通过各种总线实现层次扩展。那么按照这种层次的需要进行设备分类，一定会有全新的视角。再考虑 Webcam 软件驱动的层次，首先是功能的抽象，然后是设备连接的抽象，这正好符合设备的两个基本需求——功能和互联。设备分类完全可以从这两个层次需求的角度考虑。

3.3.1 功能型设备

这里先介绍功能型设备。讨论到功能，通常都对应到功能需求。Linux 内核直接开放一些类型的设备给用户使用，这些类型就是功能型设备。通过系统调用可以使用功能型设备。相应的框架如图 3-13 所示。

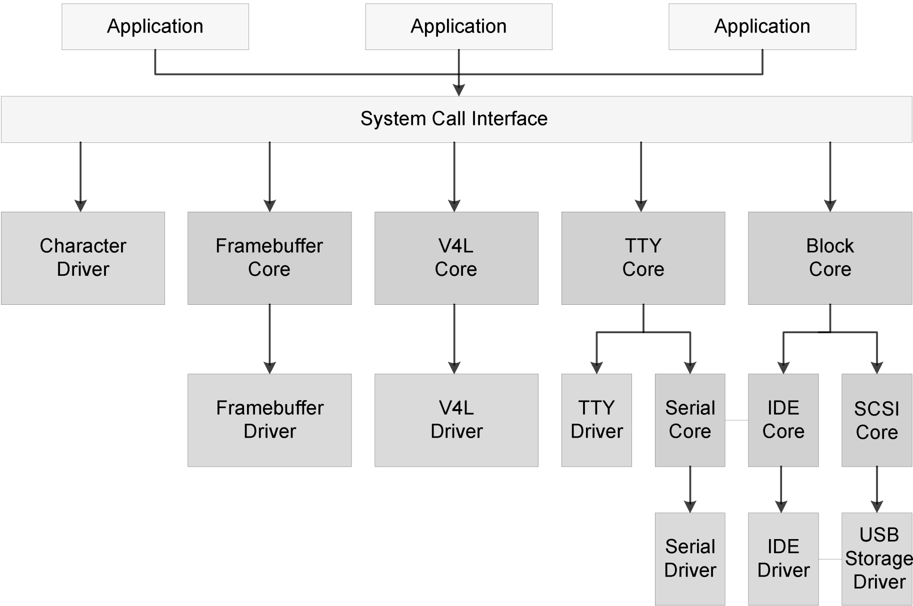


图 3-13 Linux 设备框架

从图 3-13 中可见，应用之下为一个一个具体的功能块，如 framebuffer 对应于应用图形界面的功能，V4L（video for Linux）对应于视频的功能。这些功能直接对应于应用的功能需求。还有些特殊的需求可以通过框架来进行扩展，相应的框架就是 char 设备框架和 block 设备框架。

Linux 内核提供了大部分功能设备的框架，详细的信息可以通过 proc 文件系统中的 devices 文件获得。笔者 Linux 机器上相应文件的信息如下：

```
dongfeng@DF-WS /proc $ cat devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
```

```
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 29 fb
 81 video4linux
 99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
226 drm
251 hidraw
252 bsg
253 watchdog
254 rtc

Block devices:
  1 ramdisk
  7 loop
  8 sd
  9 md
 11 sr
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
```

```
135 sd
252 device - mapper
253 virtblk
254 mdp
```

从相应的信息中可以看到，每种类型前都有一个数字，这个数字代表主设备号。可以说 Linux 内核同样考虑按功能进行设备分类，对应的方式就是为每一类功能的设备分配一个主设备号。当然 Linux 为了系统扩展，在功能的基础之上，同时考虑数据处理的特点，进而进行抽象层面的分类，这就是字符设备和块设备，也是更高层面的抽象。在功能层面可以不必考虑像字符设备和块设备这种高层次的抽象，而是应更多地关注功能框架本身。Linux 内核为这些主设备提供设备框架，使得驱动开发者的重点是在驱动框架的功能（如 input、frame buffer、v4l2 和 Alsa 等），后续章节会对这些功能设备框架进行详细介绍。对于 Linux 内核没有提供的设备框架，Linux 也提供了一定的扩展框架如 misc（Android 的 binder 就是通过 misc 实现），以及为应用层驱动提供的扩展（如 usb 和 usb_ device），当然更强大的功能扩展框架就是字符设备本身。可见 Linux 内核不仅提供了比较完善的功能设备框架，而且为功能的扩展提供了强大的支持，这些都是为了满足需求的多样性，以适应需求的变化。

3.3.2 总线型设备

总线型设备的主要功能是解决总线设备互连的问题，总线互连框架如图 3-14 所示。

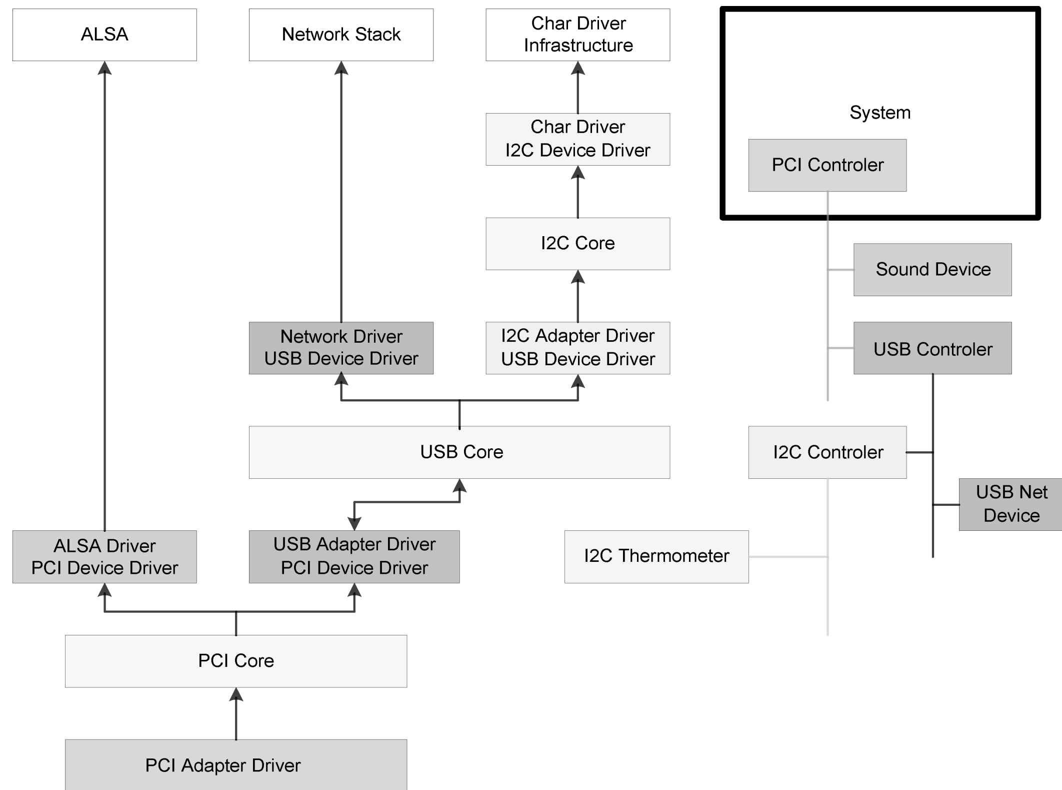


图 3-14　总线互连框架

图 3-14 的右侧是硬件连接示意图，PCI 总线上挂载音频设备和 USB 总线控制器，USB 总线上连接的设备包括 I^2C 总线控制器和 USB 网络设备，而在 I^2C 总线上则连接一个温度传感器，这是典型的总线层级连接，涉及 PCI、USB 和 I^2C 三种总线设备。图 3-14 的左侧是软件实现的框架图，可以看到，对功能驱动来说，其只见到实际设备最终连接的总线，而整个连接层次中的其他总线对功能驱动并不可见。这样做的好处就是，功能部分只关心直接连接的总线协议及其功能。什么屏蔽了不同总线连接的差异呢？总线控制器提供了该功能。总线控制器（如 USB controller）在不同的总线之间形成桥梁，并管理自己总线中连接的设备。这样具体设备就可以直接和总线控制器交互，由总线控制器屏蔽总线级联的差别。

Linux 内核同样为开发者提供了详细的总线信息，可以通过/sys/bus 获得，笔者 Linux 机器显示该目录的信息如下：

```
dongfeng@DF-WS /sys/bus $ ls -al
total 0
drwxr-xr-x 26 root root 0 Dec 10 22:05 .
dr-xr-xr-x 13 root root 0 Dec 10 22:05 ..
drwxr-xr-x  4 root root 0 Dec 10 22:05 acpi
drwxr-xr-x  4 root root 0 Dec 10 22:05 clocksource
drwxr-xr-x  4 root root 0 Dec 10 22:05 cpu
drwxr-xr-x  4 root root 0 Dec 10 22:05 event_source
drwxr-xr-x  4 root root 0 Dec 10 22:05 hid
drwxr-xr-x  4 root root 0 Dec 10 22:05 i2c
drwxr-xr-x  4 root root 0 Dec 10 22:05 machinecheck
drwxr-xr-x  4 root root 0 Dec 10 22:05 mdio_bus
drwxr-xr-x  4 root root 0 Dec 10 22:05 memory
drwxr-xr-x  4 root root 0 Dec 10 22:05 mmc
drwxr-xr-x  4 root root 0 Dec 10 22:05 node
drwxr-xr-x  5 root root 0 Dec 10 22:05 pci
drwxr-xr-x  4 root root 0 Dec 10 22:05 pci_express
drwxr-xr-x  4 root root 0 Dec 10 22:05 platform
drwxr-xr-x  4 root root 0 Dec 10 22:05 pnp
drwxr-xr-x  4 root root 0 Dec 10 22:05 rapidio
drwxr-xr-x  4 root root 0 Dec 10 22:05 scsi
drwxr-xr-x  4 root root 0 Dec 10 22:05 sdio
drwxr-xr-x  4 root root 0 Dec 10 22:05 serio
drwxr-xr-x  4 root root 0 Dec 10 22:05 spi
drwxr-xr-x  4 root root 0 Dec 10 22:05 usb
drwxr-xr-x  4 root root 0 Dec 10 22:05 virtio
drwxr-xr-x  4 root root 0 Dec 10 22:05 xen
drwxr-xr-x  4 root root 0 Dec 10 22:05 xen-backend
```

这其中包含 Linux 内核支持的各种总线，有物理的总线（如 i2c、spi、usb、pci、pci_express 等），也有虚拟总线（如 platform 等）。虚拟总线是一种逻辑总线，主要是为了满

足一种逻辑互连功能。连接的主体通常是设备和对应的驱动。总线的一个重要功能就是能够发现设备并找到合适的驱动来操作设备，这是逻辑总线的主要功能。Platform 总线主要是为 SoC 内部设备而设计的，通过该总线可以将设备属性和驱动分离，从而可以使用相同的驱动来支持同一功能核心硬件的不同设备。

通过物理总线和虚拟的逻辑总线，可以对 Linux 内核所管理的设备进行更好的组织，并且可以通过抽象分离属性和操作，给系统带来更好的扩展性。后续章节会对总线设备进行更详细的介绍。

3.4　系统实现各种无关性的框架

Linux 内核需要支持各种各样的处理器和各种各样的设备，这就要求在框架设计时能提供更多的灵活性。要实现这些，Linux 内核需要实现各种无关性的框架。其实这些无关性框架本身就是设计模式。接下来就看看 Linux 内核使用什么样的模式解决这些无关性问题。

3.4.1　体系结构无关

首先 Linux 内核需要支持各种处理器，也就是要支持各种体系结构。这就要求核心框架与是体系结构无关，另外编译系统可以简单地支持各种体系结构。

系统支持不同的体系结构，从软件管理的角度考虑，一个简单的办法就是将不同体系结构的代码放入不同的目录。Linux 内核也是如此实现的，如图 3-15 所示。

alpha	File Folder
arm	File Folder
avr32	File Folder
blackfin	File Folder
cris	File Folder
frv	File Folder
h8300	File Folder
ia64	File Folder
m32r	File Folder
m68k	File Folder
m68knommu	File Folder
microblaze	File Folder
mips	File Folder
mn10300	File Folder
parisc	File Folder
powerpc	File Folder
s390	File Folder
score	File Folder
sh	File Folder
sparc	File Folder
tile	File Folder
um	File Folder
x86	File Folder
xtensa	File Folder

图 3-15　Linux 内核体系结构相关代码目录

接下来就要在编译系统中解决相应的问题。在 Linux 内核根目录下的 Makefile 中包含如下内容：

```
# Architecture as present in compile. h
UTS_MACHINE:= $(ARCH)
SRCARCH:= $(ARCH)

# Additional ARCH settings for x86
ifeq ($(ARCH),i386)
        SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif

# Additional ARCH settings for sparc
ifeq ($(ARCH),sparc32)
        SRCARCH := sparc
endif
ifeq ($(ARCH),sparc64)
        SRCARCH := sparc
endif

# Additional ARCH settings for sh
ifeq ($(ARCH),sh64)
       SRCARCH := sh
endif

# Where to locate arch specific headers
hdr-arch   := $(SRCARCH)

ifeq ($(ARCH),m68knommu)
       hdr-arch   := m68k
endif

……

include $(srctree)/arch/$(SRCARCH)/Makefile

……

vmlinux-lds   := arch/$(SRCARCH)/kernel/vmlinux. lds

……
```

可见 Linux 内核通过宏变量 ARCH 来定义具体的体系结构，并由宏变量 SRCARCH 来定义具体的体系结构相关代码目录。首先，ARCH 转换为正确的体系结构源代码目录，然后使用体系结构相关的 Makefile 文件，这里直接引入相应目录的文件。还有一个重要的文件需要设定，这就是链接文件（由于不同的体系结构使用的地址空间会有差别）vmlinux-lds。可以看到编译不同的体系结构代码只要根据体系结构设定 ARCH 宏即可。

解决了编译的问题，就要看一下具体的实现如何做到体系结构无关了。做到体系结构无

关采用的技术就是分层，将系统分为体系结构相关层和设备控制层（将体系结构相关代码封装为统一接口）。这样设备控制层之上的代码就是体系结构无关的代码，就可以在不同的处理器上重用了。具体的实现如图 3-16 所示。

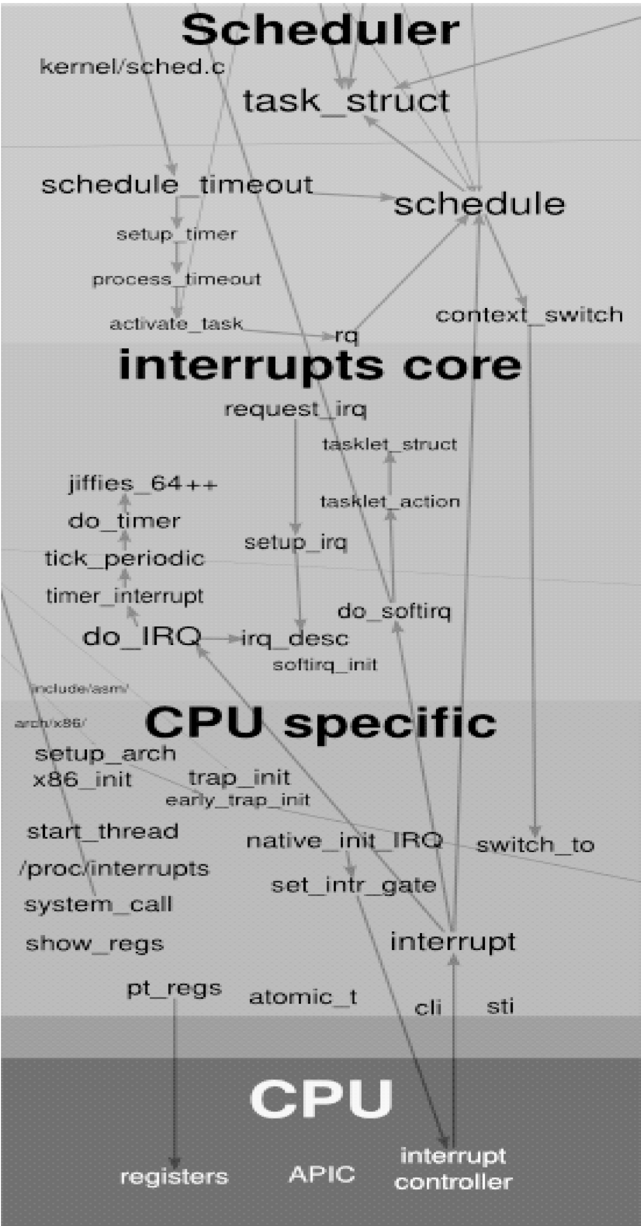


图 3-16　Linux 内核体系结构无关具体实现

从图 3-16 中可见，体系结构相关的主要部分是中断、任务切换、寄存器、系统调用以及系统初始化。只要把这几部分通过统一接口封装，就可以在代码级别实现体系结构无关。

3.4.2 功能型设备的框架与总线无关

在物理设备中，同样的总线可以连接不同的设备，同样功能的设备可以通过不同的总线使用，这样就需要软件框架中能够解决功能和总线无关的问题。下面以 SPI 总线为例说明 Linux 内核是如何解决该问题的。如图 3-17 所示。

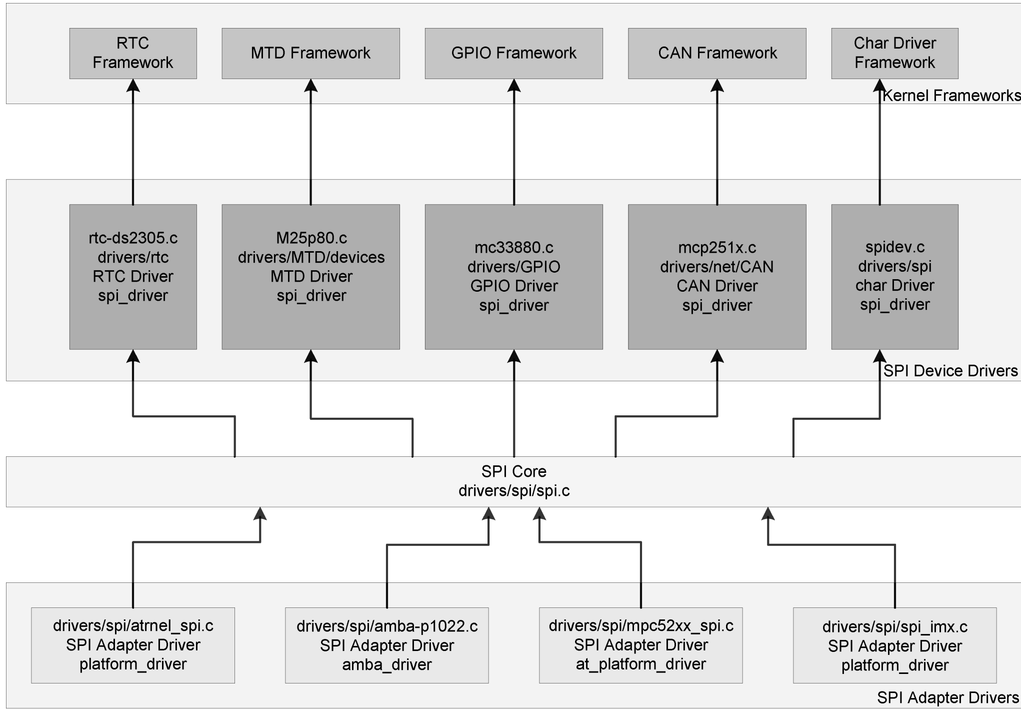


图 3-17 功能框架与 SPI 总线设备框图

图 3-17 顶部为 Linux 内核的各种框架，包括对用户的功能和横切功能；中间的 SPI Core 则是处理 SPI 总线事务，这两者是完全隔离的，它们之间的代码是无关的。当然对特定的设备需要在功能和总线操作之间建立一座桥梁，这座桥梁就是具体功能的 SPI 设备驱动。具体功能的总线驱动会在相应的功能操作和总线事务之间进行转化，从而将两个独立的框架结合起来。

通过功能框架、总线框架和特定功能的总线驱动三者的有机结合和协调工作，就可以实现功能框架和总线无关。驱动开发人员开发特定功能的总线驱动时，任务十分明确，就是在特定的功能操作和总线事务之间做转换。每个部分的功能需求明确，才会有系统性能优良、可维护性好的实现。

3.4.3 总线控制器与总线设备的无关

对总线来说，总线设备的多样性是一个方面，另一个方面处理器中对总线的管理也是多样的。比如 USB 总线，Intel 处理器的总线管理设备（即总线控制器）和 ARM 处理器的总线管理设备就有很大的差别。总线上连接的设备并不想知道自己具体和哪些总线管理设备进行交互。总线控制器只关注于具体的总线操作，所以总线控制器也并不关心连接在总线上的设

备如何使用总线实现其功能。总线控制器是一个服务管理的实体，要能为所有连接到总线上的设备进行服务，并管理它们，不至于彼此冲突。

可见总线控制器和总线设备两者要完成的任务有联系，但是彼此不能限定对方，这就要求实现总线控制器和总线设备的无关。

图 3-18 和图 3-19 分别展示了 Linux 内核在 USB 和 I^2C 总线框架上如何实现总线控制器和总线设备无关。

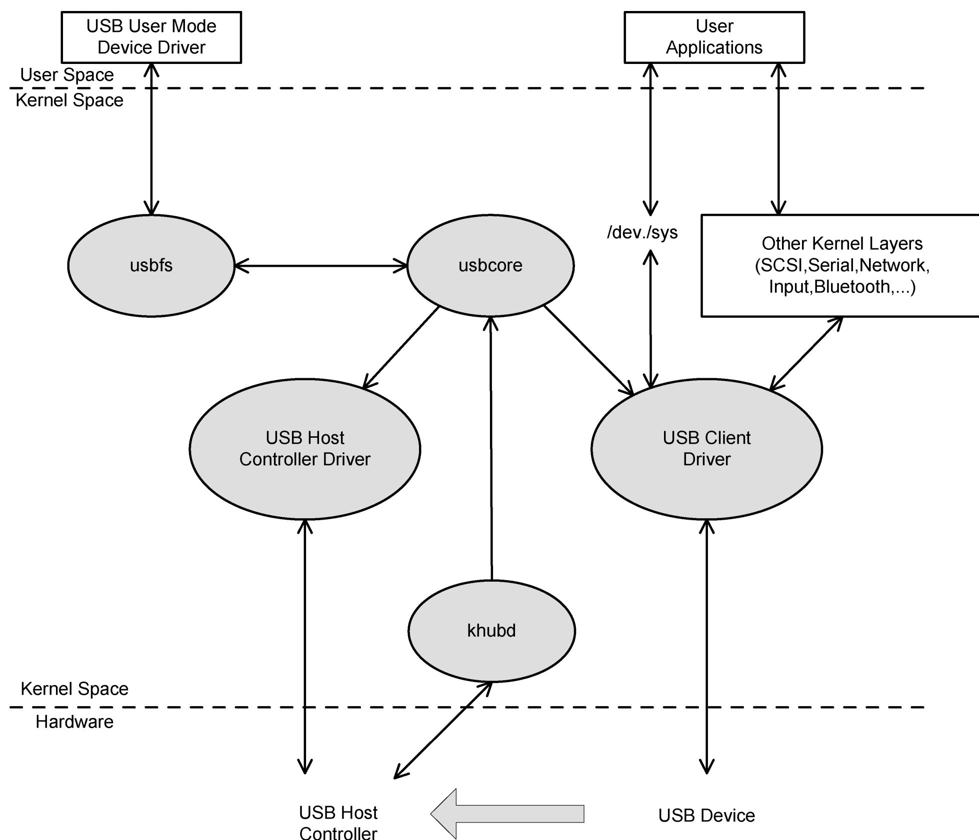


图 3-18　USB 总线框架

从图 3-18 和 3-19 中可见，不同的总线框架解决总线控制器和总线设备无关是使用了相同的方法，就是抽象出总线核心来管理总线控制器和总线设备，并将两者有机的结合。对于 USB 总线有 USB Core，对于 I^2C 总线有 I^2C Core，都是完成总线核心的功能。

可见总线核心是要同时管理总线控制器和总线设备的。如图 3-20 所示，以 USB 总线为例展示了 Linux 内核如何实现该功能。

从图 3-20 中可见，无论是总线控制器 USB adapter 还是总线设备 USB device 都会注册到 USB core 中，这样 USB core 就可以对两者进行管理。这其中的管理还包括在两者之间进行必要的绑定，比如说连接到 USB 总线控制器的 USB 设备，其操作必然要经过该总线控制器完成。这种绑定都可以通过 USB 设备发现由 USB Core 自动完成。

通过这些方法就能够实现总线控制器和总线设备无关。

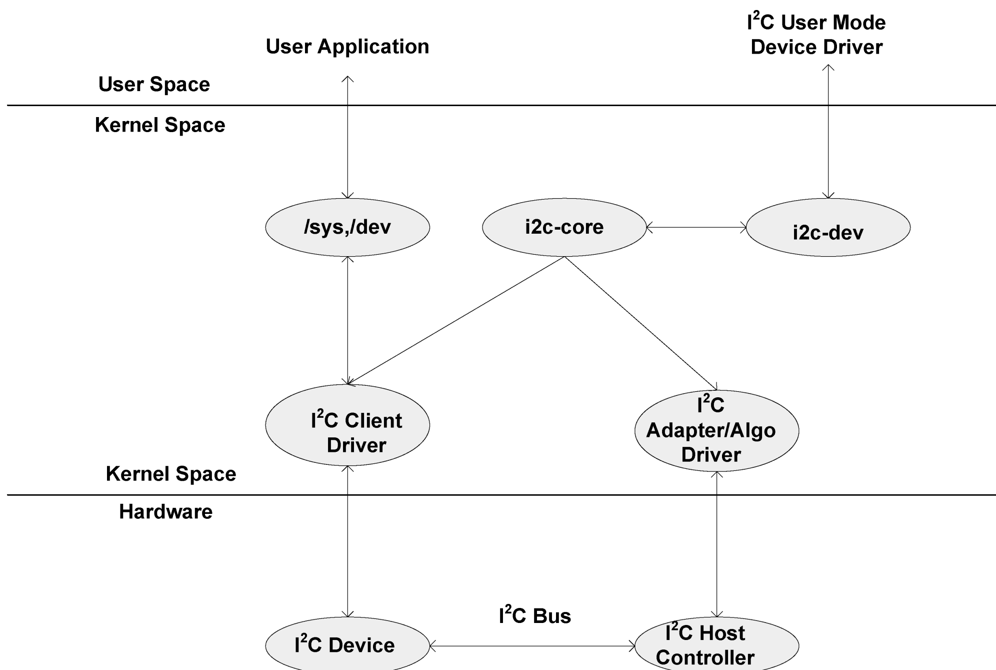


图 3-19　I²C 总线框架

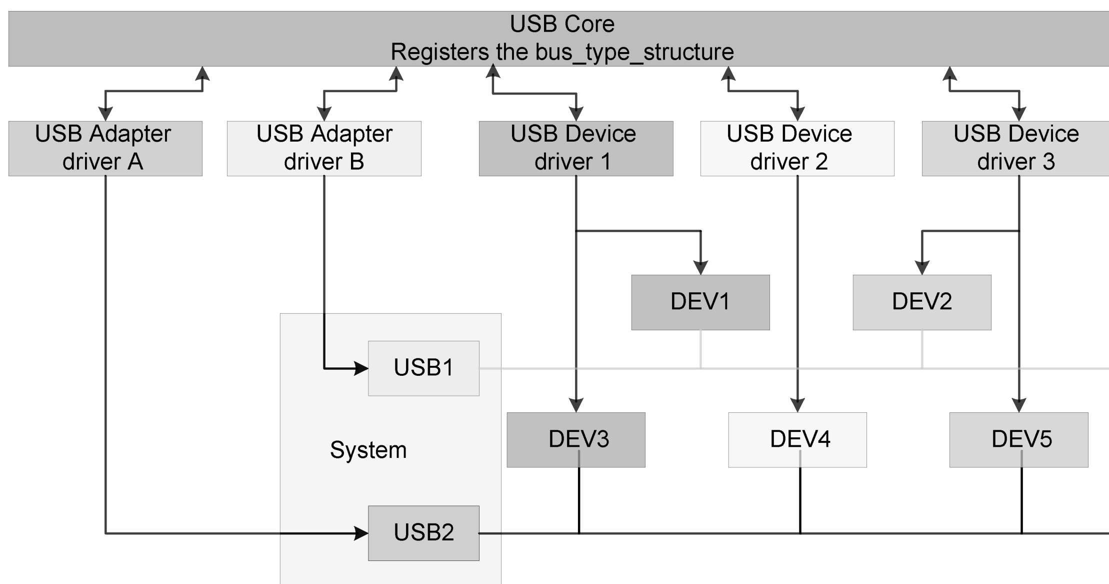


图 3-20　Linux 内核 USB 管理框架

3.4.4　设备属性和设备操作无关

图 3-20 中不仅涉及总线控制器和总线设备无关的问题，还展示了另一个问题，就是同一个设备驱动应该可以支持多个设备。如 USB device driver 1 就同时支持 DEV1 和 DEV3，这就需要使设备属性和设备操作无关。如何解决该问题呢？之前的无关性解决办

法已经给了提示，就是将需要做到无关的两部分都抽象出来，形成两个实体单独管理。对设备操作就抽象出操作接口，形成驱动的主要操作功能。将设备属性等信息抽象出来形成设备结构，不同的设备就可以有不同的属性，而作为操作接口，则通过从设备结构中获得的属性进行操作，就可以把设备属性形成的具体设备和设备操作形成的设备驱动区分开并单独进行管理。

下面是 Linux 内核中串口部分的数据结构，从中可以看出 Linux 内核是如何抽象从而将两者分离的。

代表设备的 uart_ port 中包含了各种属性信息：

```
struct uart_port {
    spinlock_t      lock;                   /* port lock */
    unsigned int    iobase;                 /* in/out[bwl] */
    unsigned char   _iomem *membase;        /* read/write[bwl] */
    unsigned int    irq;                    /* irq number */
    unsigned int    uartclk;                /* base uart clock */
    unsigned char   fifosize;               /* tx fifo size */
    unsigned char   x_char;                 /* xon/xoff flow
                                               control */
    /* ... */
};
```

代表驱动的设备操作结构，其中包含了各种操作接口，注意操作接口通常都要有设备属性结构作为参数，这样可以适用于不同的设备属性：

```
struct uart_ops {
    uint (*tx_empty)(struct uart_port *);           /* Is TX FIFO empty? */
    void (*set_mctrl)(struct uart_port *,
                      unsigned int mctrl);          /* Set modem control params */
    uint (*get_mctrl)(struct uart_port *);          /* Get modem control params */
    void (*stop_tx)(struct uart_port *);            /* Stop xmission */
    void (*start_tx)(struct uart_port *);           /* Start xmission */
    /* ... */
    void (*shutdown)(struct uart_port *);           /* Disable the port */
    void (*set_termios)(struct uart_port *,
                        struct termios *new,
                        struct termios *old);       /* Set terminal interface
                                                       params */
    /* ... */
    void (*config_port)(struct uart_port *,
                        int);                       /* Configure UART port */
    /* ... */
};
```

具体两者绑定操作通常在总线级别完成。没有物理总线的设备可以通过逻辑总线（如platform 总线）完成。

可见要实现设备属性和设备操作无关，不仅是设备和驱动之间的问题，还要涉及高级别的抽象即总线，通过总线将两者互连绑定。这部分可以扩展到具体的驱动以及 Linux 内核中的设备模型相关的内容，后续会有更详细的说明。

3.4.5 策略和机制无关

现实生活中会有不同的人使用相同的工具解决同一个问题。由于人的不同，解决问题的方法就不同，即使使用相同的工具，具体的操作方式也会有差别。在系统设计方面，由于系统是十分复杂的，要解决一个问题时，不同的情况下采用不同的方法会有不同的效果，这和现实生活中使用工具解决问题是类似的。在 Linux 内核中也面临同样的问题，只是把基本的工具理解为机制，而具体操作工具的方式方法理解为策略问题。这样系统就要实现策略和机制无关。

图 3-21 是 Linux 内核中 CPUFreq 的架构，其中涉及到该架构中如何解决策略和机制的方法。

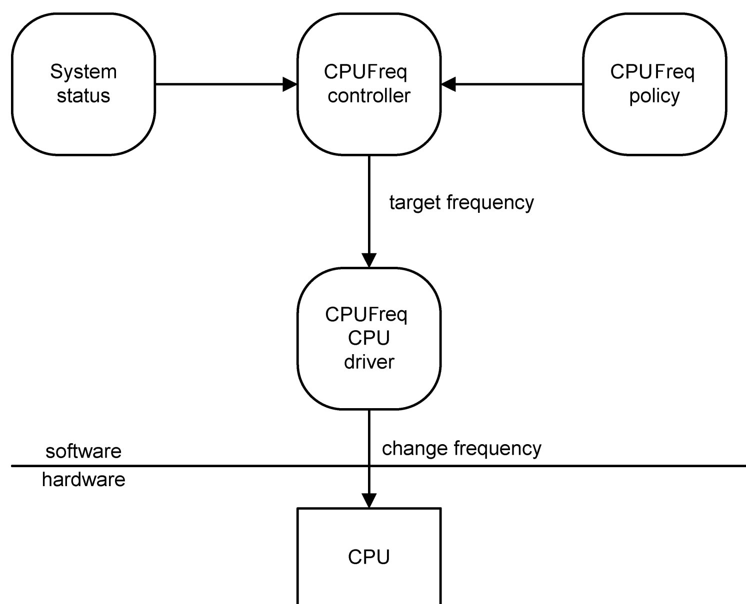


图 3-21　CPUFreq 架构

图 3-21 中 CPU driver 代表了机制，而 CPUFreq policy 则是具体进行 CPU 主频选择的策略，CPUFreq controller 根据当前系统的状态和当前选择的策略选择合适的 CPU 主频，通过 CPU driver 进行设置。从其过程可以看出，机制（即对 CPU 设置一定的频率）和策略（什么情况下选择什么频率）已经分离。这样就可以通过相同的机制支持不同的策略，通过对策略的选择，使得系统随需而变。

具体的机制和策略无关，Linux 内核中还有很多体现，后续会有更详细的说明。

3.5 内核提供的基本服务和接口简介

Linux 内核的使用环境和应用层的程序完全不同，主要在于有很多资源的限制，如栈容量的限制，内存不一定是连续物理空间的限制等等。由于这些限制，应用层的库函数就不能直接在 Linux 内核中使用，但是有很多功能确实是 Linux 内核中各个模块都需要的。这就需要 Linux 内核提供类似 C 库功能的函数库，以适应内核有限的资源，并为 Linux 内核的其他模块使用。进行 Linux 内核和驱动程序的开发自然需要熟悉这些基本服务和接口。下面具体介绍 kernel API。

3.5.1 基本数据类型

1. 双向链表

双向链表的数据结构如下：

```
struct list_head {
    struct list_head *next, *prev;
};
```

该双向链表要嵌入到具体的结构中使用。其优点是操作与所嵌入的结构无关。由于其嵌入到具体的结构中，所以需要通过双向链表查找到具体的包含双向链表的结构体指针，该操作通过宏 container_of 完成，具体如下：

```
#define container_of(ptr, type, member) ({                    \
    const typeof(((type *)0)->member) *__mptr = (ptr);    \
    (type *)((char *)__mptr - offsetof(type,member));})
```

对双向链表的具体操作如下：

```
list_add ——向链表添加一个条目
list_add_tail ——添加一个条目到链表尾部
__list_del_entry ——从链表中删除相应的条目
list_replace——用新条目替换旧条目
list_del_init——从链表中删除条目后重新初始化
list_move——从一个链表中删除并加入为另一个链表的头部
list_move_tail——从一个列表中删除并加入为另一个链表的尾部
list_is_last——测试是否为链表的最后一个条目
list_empty——测试链表是否为空
list_empty_careful——测试链表是否为空并没有被修改
list_rotate_left——向左转动链表
list_is_singular——测试链表是否只有一个条目
list_cut_position——将链表一分为二
list_splice——将两个链表进行合并
```

```
list_splice_tail——将两个链表进行合并为一个链表
list_splice_init——将两个链表进行合并为一个链表并初始化为空表
list_splice_tail_init——将两个链表进行合并为一个链表(从尾部合并)并初始化为空表
list_entry——获取条目的结构,实现对 container_of 的封装
list_first_entry——获取链表的第一个元素
list_first_entry_or_null——获取链表的第一个元素
list_for_each——遍历链表
list_for_each_prev——反向遍历链表
list_for_each_safe——遍历链表并删除链表中相应的条目
list_for_each_prev_safe——反向遍历链表并删除链表中相应的条目
list_for_each_entry——遍历指定类型的链表
list_for_each_entry_reverse——反向遍历指定类型的链表
list_prepare_entry——准备一个用于 list_for_each_entry_continue 的条目
list_for_each_entry_continue——从指定点开始继续遍历指定类型的链表
list_for_each_entry_continue_reverse——从指定点开始反向遍历链表
list_for_each_entry_from——从当前点遍历指定类型的链表
list_for_each_entry_safe——反向遍历指定类型的链表并删除链表中相应的条目
list_for_each_entry_safe_continue——继续遍历链表并删除链表中相应的条目
list_for_each_entry_safe_from——从当前点遍历链表并删除链表中相应的条目
list_for_each_entry_safe_reverse——反向遍历链表并删除链表中相应的条目
list_safe_reset_next——获得下一个指定类型的条目
hlist_for_each_entry——遍历指定类型的单指针表头链表
hlist_for_each_entry_continue——从当前点继续遍历单指针表头链表
hlist_for_each_entry_from——从当前点继续遍历单指针表头链表
hlist_for_each_entry_safe——遍历指定类型的单指针表头链表并删除链表中相应的条目
```

2. 字符串相关

内核中经常会有字符串转换的需要，其接口如下：

```
simple_strtoull——变换一个字符串为无符号的 long long 型
simple_strtoul——变换一个字符串为无符号的 long 型
simple_strtol——变换一个字符串为有符号的 long 型
simple_strtoll——变换一个字符串为有符号的 long long 型
vsnprintf——格式化一个字符串并放入缓冲区
vscnprintf——格式化一个字符串并放入缓冲区
snprintf——格式化一个字符串并放入缓冲区
scnprintf——格式化一个字符串并放入缓冲区
vsprintf——格式化一个字符串并放入缓冲区
sprintf——格式化一个字符串并放入缓冲区
vbin_printf——解析格式化字符串并将二进制值放入缓冲区
bstr_printf——对二进制参数进行格式化字符串操作并放入缓冲区
bprintf——解析格式化字符串并将二进制值放入缓冲区
```

vsscanf——从格式化字符串中分离出的参数列表
sscanf——从格式化字符串中分离出的参数列表
kstrtol——变换一个字符串为 long 型
kstrtoul——变换一个字符串为无符号的 long 型
kstrtoull——变换一个字符串为无符号的 long long 型
kstrtoll——变换一个字符串为 long long 型
kstrtouint——变换一个字符串为无符号的 int 型
kstrtoint——变换一个字符串为 int 型

另外字符串本身的操作接口如下：

strnicmp——长度有限的字符串比较,这里不分大小写
strcpy——复制一个以 NULL 结尾的字符串
strncpy——复制一个以 NULL 结尾的有限长度字符串
strlcpy——复制一个以 NULL 结尾的有限长度字符串到缓冲区中
strcat——在字符串后附加以 NULL 结尾的字符串
strncat——在字符串后附加以 NULL 结尾的一定长度的字符串
strlcat——在字符串后附加以 NULL 结尾的一定长度的字符串
strcmp——比较两个字符串
strncmp——比较两个限定长度的字符串
strchr——在字符串中查找第一个出现指定字符的位置
strrchr——在字符串中查找最后出现指定字符的位置
strnchr——在字符串中查找出现指定字符串的位置
skip_spaces——从字符串中移除前置空格
strim——从字符串中移除前置及后置的空格
strlen——获得字符串的长度
strnlen——获得一个有限长度字符串的长度
strspn——计算一个仅包含可接受字母集合的字符串的长度
strcspn——计算一个不包含指定字母集合的字符串的长度
strpbrk——找到字符集合在字符串第一次出现的位置
strsep——分割字符串
sysfs_streq——字符串比较,用于 sysfs
strtobool——用户输入转换成布尔值
memset——内存填充
memcpy——内存复制
memmove——内存复制
memcmp——内存比较
memscan——在内存中找指定的字符
strstr——在一个以 NULL 结尾的字符串中找到第一个子串
strnstr——在一个限定长度字符串中找到第一个子串
memchr——找到内存中的字符
memchr_inv——找到内存中的不匹配字符

3. 位操作

内核中很多地方需要位操作，比如磁盘管理等对空间管理的功能。

```
set_bit——设置内存中指定位
__set_bit——设置内存中指定位
clear_bit——清除内存中指定位
__change_bit——改变内存中指定位
change_bit——改变内存中指定位
test_and_set_bit——设置内存中指定位并返回旧值
test_and_set_bit_lock——设置内存中指定位并返回旧值
__test_and_set_bit——设置内存中指定位并返回旧值
test_and_clear_bit——清除内存中指定位并返回旧值
__test_and_clear_bit——清除内存中指定位并返回旧值
test_and_change_bit——改变内存中指定位并返回旧值
test_bit——测试内存中某一位是否设置
__ffs——找到 word 里的第一个设置位
ffz——找到 word 里的第一个零位
ffs——找到 word 里的第一个设置位
fls——找到 word 里的最后一个设置位
fls64——找到 64 位 word 里的最后一个设置位
```

3.5.2 基本原子操作

处理器中很多单元都可以对 RAM 进行操作，这就需要考虑并发访问的问题，特别是一些变量的操作，需要有原子操作的接口。Linux 内核中这样的操作也是很多的，而且 Linux 内核还需要面对多体系结构的问题。为了在这方面简化开发人员的操作，Linux 内核提供了体系结构无关的原子操作接口。

```
atomic_read——读原子量
atomic_set——设置原子量
atomic_add——在原子量中增加整数
atomic_sub——从原子量中减去整数
atomic_sub_and_test——从原子量中减去整数并测试结果
atomic_inc——增加原子量
atomic_dec——减少原子量
atomic_dec_and_test——减少并测试
atomic_inc_and_test——增加并测试
atomic_add_negative——增加并测试是否为负
atomic_add_return——增加整数并返回
atomic_sub_return——减少整数并返回
__atomic_add_unless——增加到给定值为止
atomic_inc_short——增加一个短整数
atomic_or_long——或两个长整数
```

3.5.3 延时、调度、定时器相关

Linux 内核通常会对某些操作等待一定的时间后，再确认操作完成。根据系统的需要这种延时操作有不同的实现方式，主要包括延时、调度和定时器等几种不同的方法。有时还要检查系统所处的状态，具体的接口如下：

pid_alive——检查一个进程是否过期
is_global_init——检查一个进程的结构体是否初始化
is_idle_task——一个指定的进程是否空闲
threadgroup_lock——锁定线程组
threadgroup_unlock——解锁线程组
wake_up_process——唤醒一个特定的进程
preempt_notifier_register——抢占或者调度事件的通知注册接口
preempt_notifier_unregister——取消接收抢占或者调度事件的通知
__wake_up——唤醒等待队列中挂起的线程
__wake_up_sync_key——唤醒等待队列中挂起的线程
complete——对等待完成量的单一线程发信号
complete_all——对等待完成量的所有线程发信号
wait_for_completion——在完成量上等待直到其处于完成状态
wait_for_completion_timeout——在完成量上等待直到其处于完成状态,允许超时退出
wait_for_completion_io——在完成量上等待直到其处于完成状态
wait_for_completion_io_timeout——在完成量上等待直到其处于完成状态,允许超时退出
wait_for_completion_interruptible——在完成量上等待直到其处于完成状态,允许信号中断
wait_for_completion_interruptible_timeout——在完成量上等待直到其处于完成状态
wait_for_completion_killable——在完成量上等待直到其处于完成状态,允许杀死该进程
wait_for_completion_killable_timeout——在完成量上等待直到其处于完成状态
try_wait_for_completion——尝试对完成量进行非阻塞的减少操作
completion_done——检查是否有进程等待完成量
task_nice——返回给定任务的调度 nice 值
sched_setscheduler——改变一个线程的调度策略或优先级
yield——放弃处理器,触发调度
yield_to——放弃处理器给同线程组中的其他线程
cpupri_find——找到系统中最好的(最低优先级)的处理器
cpupri_set——更新处理器优先级设置
cpupri_init——初始化处理器优先级结构
cpupri_cleanup——清除处理器优先级结构
get_sd_load_idx——获取给定进程调度域的负载指数
update_sg_lb_stats——更新进程调度组的负载均衡统计
update_sd_pick_busiest——最忙的组返回真
update_sd_lb_stats——更新进程调度域的负载均衡统计
check_asym_packing——检查此组是否能加入相应的调度域
fix_small_imbalance——在负载均衡阶段计算一个进程调度域中各组间存在的次要不平衡

calculate_imbalance——在负载均衡阶段计算当前进程调度域中各组目前不平衡数目
find_busiest_group——如有不平衡返回进程调度域中最忙的一个组
DECLARE_COMPLETION——声明并初始化一个完成量结构
DECLARE_COMPLETION_ONSTACK——声明并初始化一个完成量结构
init_completion——初始化一个动态分配的完成量
INIT_COMPLETION——重新初始化一个完成量结构
__round_jiffies——返回精确到秒的 jiffies 值
__round_jiffies_relative——返回精确到秒的 jiffies 值
round_jiffies——返回精确到秒的 jiffies 值
round_jiffies_relative——返回精确到秒的 jiffies 值
__round_jiffies_up——返回精确到秒的 jiffies 值,返回值大于原值
__round_jiffies_up_relative——返回精确到秒的 jiffies 值,返回值大于原值
round_jiffies_up——返回精确到秒的 jiffies 值,返回值大于原值
round_jiffies_up_relative——返回精确到秒的 jiffies 值,返回值大于原值
set_timer_slack——设置定时器允许的偏差
init_timer_key——初始化一个定时器
mod_timer_pending——修改一个挂起定时器的超时时间
mod_timer——修改一个定时器的超时时间
mod_timer_pinned——修改一个定时器的超时时间
add_timer——开始一个定时器
add_timer_on——在特定的处理器上开始一个定时器
del_timer——关闭一个定时器
try_to_del_timer_sync——尝试关闭一个定时器
del_timer_sync——关闭一个定时器并等待处理程序的完成
schedule_timeout——休眠直到超时
msleep——毫秒级休眠
msleep_interruptible——毫秒级休眠可以被信号唤醒
usleep_range——微秒级休眠

3.5.4 锁操作

作为复杂系统，不同模块之间一定有公共操作部分。对公共部分的访问是需要锁保护的。Linux 内核中有上下文的概念，包括中断上下文和进程上下文，这两个上下文之间对共同数据的访问同样需要保护。另外现代处理器中通常包含多核（SMP），不同的核很可能访问相同的数据，这样也需要保护。Linux 内核为了支持不同形式的数据保护提供了各种锁机制。

针对 SMP、中断上下文、进程上下文的数据访问保护，Linux 内核提供了自旋锁相关的锁机制，主要接口如下：

spin_lock——获取指定的自旋锁。
spin_lock_irq ——禁止本地中断并获取指定的锁。
spin_lock_irqsave ——保存本地中断的当前状态,禁止本地中断,并获取指定的锁。
spin_unlock——释放指定的锁。

```
spin_unlock_irq ——释放指定的锁,并激活本地中断。
spin_unlock_irqstore ——释放指定的锁,并让本地中断恢复到以前状态。
spin_lock_init ——动态初始化指定的 spinlock_t。
spin_trylock ——试图获取指定的锁,如果未获取,则返回 0。
spin_is_locked ——如果指定的锁当前正在被获取,则返回非 0,否则返回 0。
```

对不同进程上下文之间的数据访问保护，为了保证性能，Linux 内核根据读写操作的频繁程度不同提供了各种锁机制，如读写锁、RCU 等（主要应用于文件系统和网络部分）。但是驱动开发等最常使用的还是 mutex（互斥）操作，相应的操作接口如下：

```
mutex_init——初始化互斥量。
mutex_is_locked——检查互斥量是否被锁定。
mutex_lock——获取互斥量。
mutex_unlock——释放互斥量。
ww_mutex_unlock——释放 W/W 互斥量。
mutex_lock_interruptible——获取互斥量可以被信号中断。
mutex_trylock——尝试非阻塞获取互斥量。
atomic_dec_and_mutex_lock——原子减操作,当为 0 时获得互斥量。
```

这两种锁的使用方法见表 3-1。

表 3-1　自旋锁和互斥使用方法

需　　求	建议的加锁方法
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期加锁	优先使用互斥体
中断上下文中加锁	使用自旋锁并要关中断
持有锁并需要睡眠	使用互斥体

内核中的情况很复杂，保护数据的互斥及一致性操作也有很多种，以上只是简单介绍了两种，后面会有更详细的介绍。

3.5.5　抢占和屏障

从 2.6 版本之后的内核已经是抢占式内核，也就是说在内核态就可以实现抢占。这对内核来说是质的飞跃，可以提升系统实时性与整个系统的效率，同时也给系统开发增加了难度。某些特殊情况需要禁止和使能内核抢占。

其实使用自旋锁已经可以防止内核抢占了，但是有时候仅仅需要禁止内核抢占，不需要像自旋锁那样消耗大的锁。这时候就需要使用禁止内核抢占的方法，具体如下：

```
preempt_disable ——增加抢占计数，从而禁止内核抢占。
preempt_enable ——减少抢占计数，并当该值降为 0 时检查和执行被挂起的需调度的任务。
preempt_enable_no_resched ——激活内核抢占但不再检查任何被挂起的需调度的任务。
preempt_count ——返回抢占计数。
```

随着技术的发展，编译器和处理器会对代码进行很多优化，现如今处理器还可以进行乱序执行。一段代码由于编译器或者处理器在编译和执行时对执行顺序进行优化，会使代码的执行顺序和所写的代码有区别。一般情况下，这没有什么问题，但是在并发条件下，可能会出现取得的值与预期不一致的情况。在某些并发情况下，为了保证代码的执行顺序，引入了一系列屏障方法来阻止编译器和处理器的优化，具体方法如下：

```
rmb——阻止跨越屏障的载入动作发生重排序。
read_barrier_depends——阻止跨越屏障的具有数据依赖关系的载入动作重排序。
wmb——阻止跨越屏障的存储动作发生重排序。
mb——阻止跨越屏障的载入和存储动作重新排序。
smp_rmb——在 SMP 上提供 rmb()功能，在 UP 上提供 barrier()功能。
smp_read_barrier_depends——在 SMP 上提供 read_barrier_depends()功能，在 UP 上提供 barrier()功能。
smp_wmb——在 SMP 上提供 wmb()功能，在 UP 上提供 barrier()功能。
smp_mb——在 SMP 上提供 mb()功能，在 UP 上提供 barrier()功能。
barrier——阻止编译器跨越屏障对载入或存储操作进行优化。
```

3.6 小结

本章首先对 Linux 内核框架进行层次分析，然后从需求的角度探讨 Linux 内核的功能及设备分类，接着探讨 Linux 内核需要解决的无关性问题以及实现方法，最后介绍 Linux 内核提供的基本库功能。

Linux 内核是不断发展的，也是不断变化的，具体都是为了适应需求的变化。对 Linux 内核的学习和研究要从需求的角度出发，采用层次化分析的方法，重点关注其采用何种设计模式解决问题，并要细化到具体的 API，这样才能从整体到细节把握好 Linux 内核开发，成为 Linux 内核和驱动开发达人。

第 4 章　内核核心介绍及硬件的具体实现

关于 Linux 内核核心，第 3 章已经简单分析了需求和功能，接下来就对 Linux 内核核心各个部分进行详细的介绍和分析。主要集中在芯片相关的功能模块的介绍和分析。在层次上本章的内容属于硬件接口层、逻辑层等底层的功能模块。详细了解相关内容对芯片移植相关的工作会有很大的帮助。

4.1　内核初始化

初始化是进入系统首先执行的部分，可以说是系统的入口，其中会涉及系统中各个模块的初始化，对内核来说也不例外。内核的初始化不同于其他系统，有其自身的特殊性，比如和处理器相关性比较大，关系到整个系统的正常运行等。注意：Linux 内核初始化，不仅要使硬件和其自身进入正确的执行状态，还要使得应用系统进入合适的状态。从这些角度看理解初始化对了解 Linux 内核及整个系统有十分重要的意义。

4.1.1　内核初始化的基本需求

Linux 系统中内核处于硬件和应用层之间。整个系统启动和初始化的过程，Linux 内核是在主处理器启动之后才会执行。不同的处理器启动流程并不相同，这就要求 Linux 内核能支持各种处理器的初始化操作。Linux 内核各个模块，大部分设计时做到了体系结构无关。这就要求在初始化过程中要将体系结构相关和无关部分分开。另外同一体系结构，由于外设的不同也会有板级的差别，在初始化的框架中也要考虑板级的接口。内核相关的初始化既要有体系结构相关的部分，又要有板级相关的部分。

Linux 内核是一个庞大的系统，最终生成的操作系统代码的执行文件非常大。如果直接使用执行文件，就要求外部存储保留很大的空间用于存放操作系统的执行镜像。启动过程读取执行镜像文件比较长，这对于系统使用来说是不经济的。考虑到系统在内存中执行速度会很快，而启动时如果将比较小的压缩文件读入内存，然后解压，这样就会节省外部存储的空间。另外现在的处理器从内存解压比从外部存储读取速度快得多，所以通常在内存中解压启动也会快，这就需要 Linux 内核初始化的时候，能够将压缩的 image 进行解压，当然解压功能应该由 Linux 内核提供。

普通应用一般都是有初始化参数的，当然内核同样需要有初始化参数。内核中各个模块也有这种需求，这就要求内核能提供一种方式进行初始化参数设置，并且能标定不同模块的初始化参数，还要将初始化参数传给对应的模块。

还有一个需求是和内存占用相关的，通常初始化代码只在初始化的时候执行，而在系统的运行过程中并不需要，这样在系统初始化之后就应该把这部分内存释放以减少这部分内存资源的无故浪费。

4.1.2 内核初始化框架介绍

谈到内核初始化，首先要考虑到处理器的初始化过程。处理器上电后，内存（可以执行代码的设备）还没有初始化、没有处于正常的工作状态，也就不能直接执行代码。对于该问题，各种体系结构都会规定上电重启（通常为 power on reset）的情况下，处理器执行第一条指令的地址，而具体的处理器厂商就将可执行代码存放在该地址处，使得处理器上电后能正常运行，通常这部分代码被叫做 ROM Code。处理器首先执行 ROM Code 的代码，处理器中还有一部分存储空间即内部 SRAM 可以执行代码，但是 SRAM 通常都比较小，不能放下 Linux 内核的压缩 image，这样就需要有个中间过渡的模块，该模块的主要功能是进行系统加载，叫做 boot loader。当然根据具体的情况，boot loader 也可以分级，最终 boot loader 会加载 Linux 内核的压缩 image，并将参数传给 Linux 内核，跳转到 Linux 内核中让其自身初始化并执行。这个阶段的流程如图 4-1 所示。

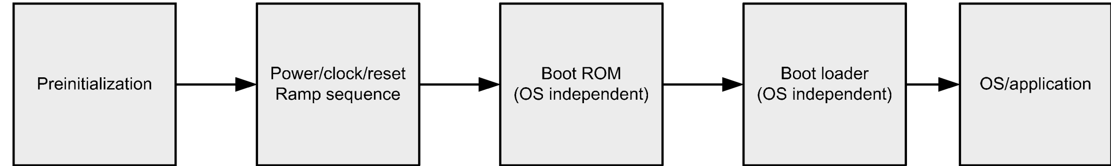


图 4-1　处理器启动流程

图 4-1 引自《DM 3730 芯片手册》中第 3524 页的流程图，虽然是 DM 3730 芯片手册中的图，但是该流程广泛适用于各种处理器。

Rom Code 通常是由处理器厂商进行开发，而所谓的软件初始化是从 boot loader 开始的。整个软件系统初始化流程如图 4-2 所示。

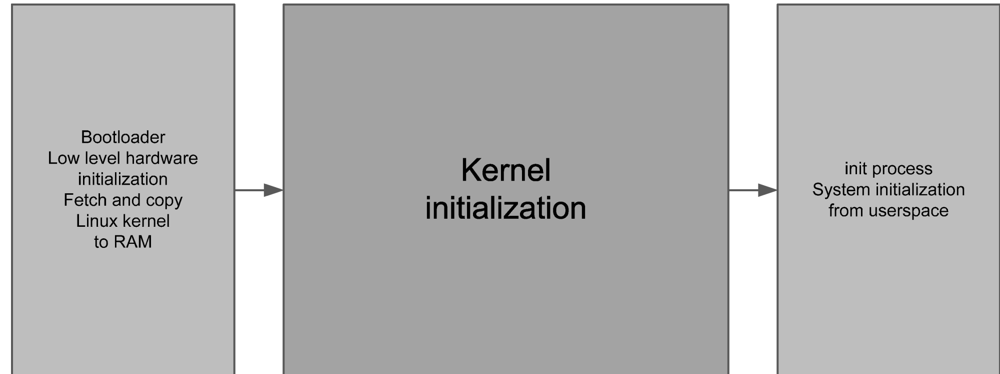


图 4-2　系统初始化流程

由图 4-2 可见，boot loader 的主要工作包括两部分：其一，进行一些硬件的初始化，如内存以及一些必要设备的初始化；另外，boot loader 还要获得 Linux 内核 image 并将其复制到内存中。但是注意 boot loader 能使用的资源比较有限，所以功能比较简单。内核初始化的主要任务则是加载并执行应用空间的初始化程序，由系统应用接管系统完成特定的功能，而 Linux 内核则开始为应用提供良好的服务。

Linux 内核初始化框架如图 4-3 所示。

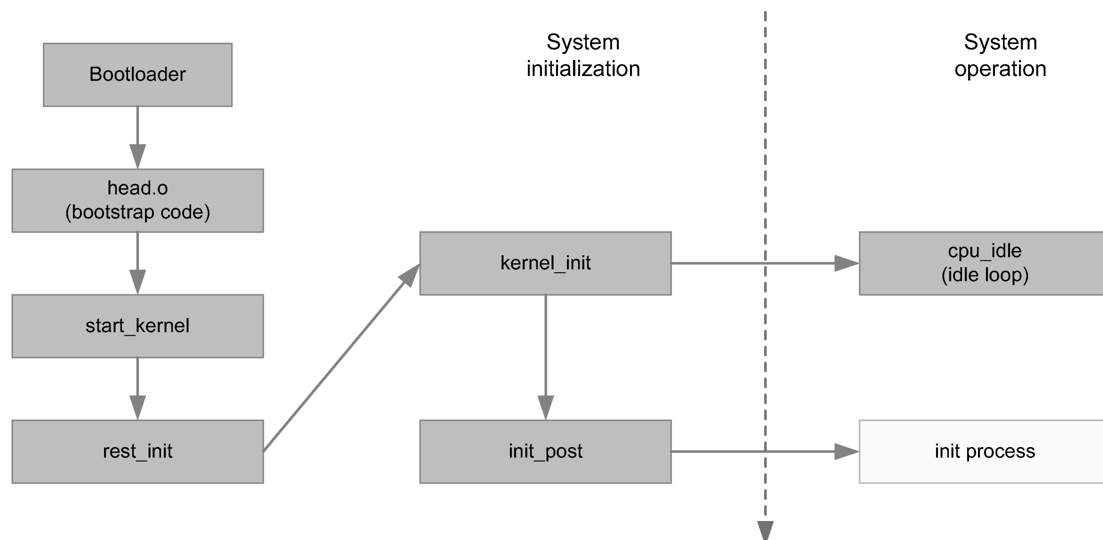


图 4-3　Linux 内核初始化框架

图 4-3 展示了 Linux 内核的初始化的基本流程，首先是与体系结构相关的 head. o 部分，然后跳入体系结构无关的代码 start_ kernel（这里已经进入虚拟地址执行），开始内核后续的初始化处理。其中会涉及很多模块的初始化，当然有些模块中包含体系结构相关以及板级相关的初始化操作。最终 Linux 内核会执行应用系统的初始化进程 init process，从而真正地转入应用系统中，开始各种各样应用的操作。

4. 1. 3　TI 芯片内核初始化相关实现详解

下面以 4. 1. 1 节所介绍的系统初始化流程为例，来看看不同的处理器启动流程的区别。图 4-4 和图 4-5 分别是 DM 816X 和 DM 3730 的启动流程。

从图 4-4 和图 4-5 可见，DM 816X 和 DM 3730 启动流程还是有区别的。主要区别在于：DM 816X 的 boot loader 阶段只是 u-boot，而 DM 3730 的 boot loader 阶段则分为 x-loader 和 u-boot。其中原因是芯片内部片内内存大小。DM 3730 中只有 64 KB 的片内内存，而 DM 816X 有 512 KB，这样 DM 816X 就可以直接将 u-boot 的 image 放入片内内存中，而 DM 3730 的片内内存空间不够，就需要通过一个 mini boot loader 来过渡。可见资源对于系统的设计还是有很大影响的。

接下来以 TI 芯片为基础进行整个系统的初始化过程详解。从这个过程中可以了解到系统移植的信息。

1. image 文件生成

Linux 内核的编译过程，首先会将各个模块单独编译，然后链接并最终生成一个很大的文件，这个文件就是 vmlinux。其中当然包括初始化的代码，而整个 Linux 系统的起始地址这一重要信息在 vmlinux. lds 内（此文件是体系结构相关的）。以 ARM 为例，该文件是由 arch/arm/kernel/vmlinux. lds. S 在编译链接时生成。下面先看看 DM 3730 编译链接生成的该文件的部分内容。

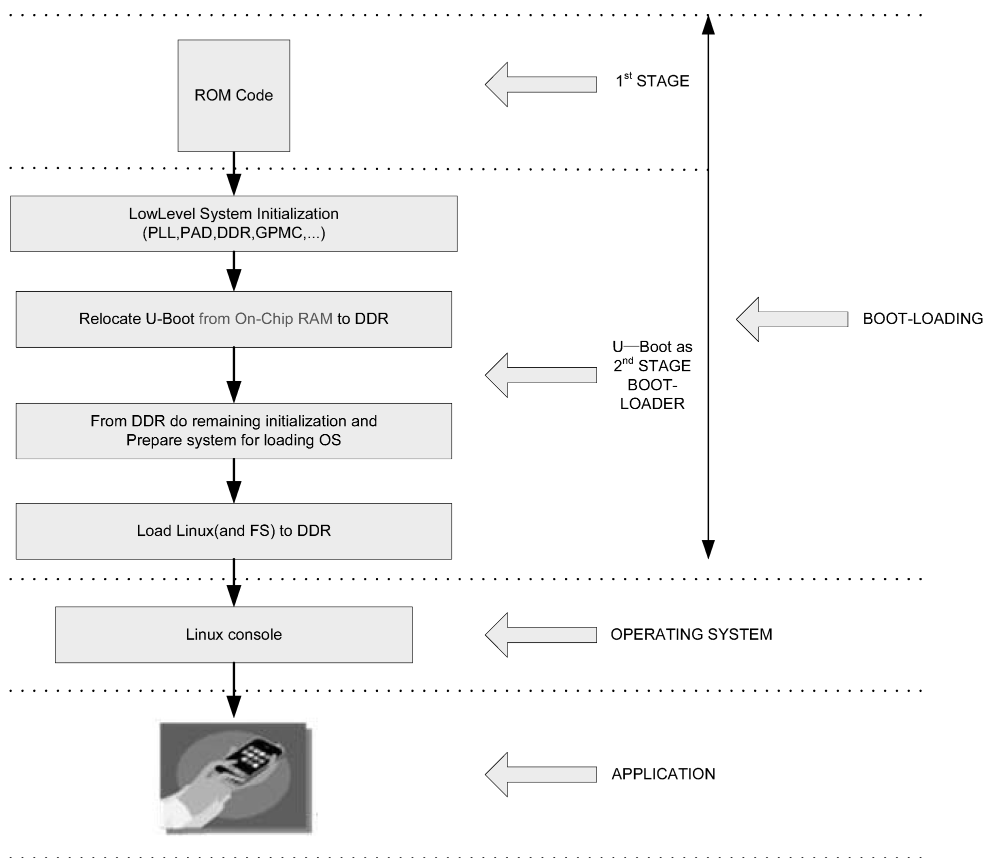


图 4-4　DM 816X 启动流程

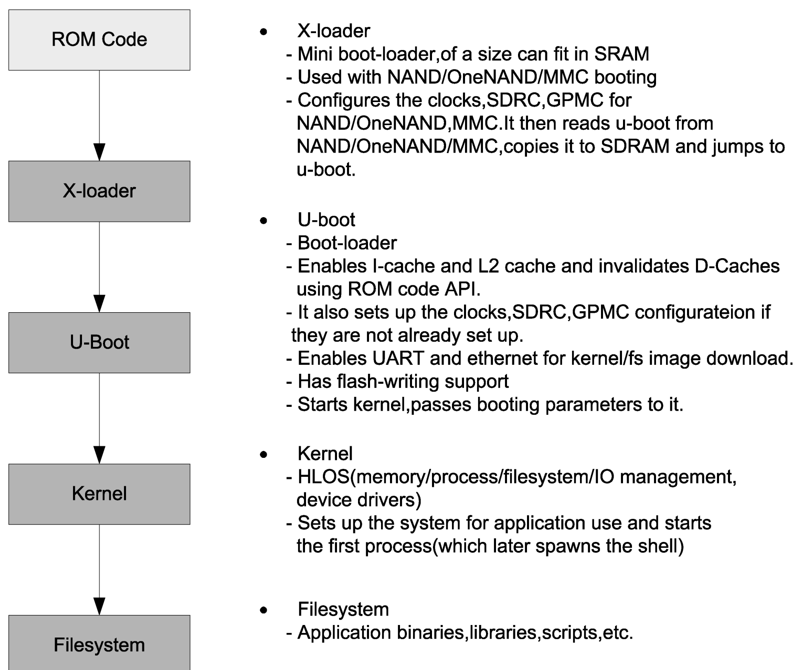


图 4-5　DM 3730 启动流程

```
OUTPUT_ARCH(arm)
ENTRY(stext)
jiffies = jiffies_64;
SECTIONS
{
  . = 0xC0000000 + 0x00008000;
  .init : { /* Init code and data */
  _stext = .;
  _sinittext = .;
   *(.head.text)
   *(.init.text) *(.cpuinit.text) *(.meminit.text)
  _einittext = .;
  __proc_info_begin = .; *(.proc.info.init) __proc_info_end = .;
  __arch_info_begin = .;
   *(.arch.info.init)
  __arch_info_end = .;
  __tagtable_begin = .;
   *(.taglist.init)
  __tagtable_end = .;
  . = ALIGN(16); __setup_start = .; *(.init.setup) __setup_end = .;
  __initcall_start = .; *(.initcallearly.init) __early_initcall_end = .; *(.initcall0.init) *
(.initcall0s.init) *(.initcall1.init) *(.initcall1s.init) *(.initcall2.init) *(.initcall2s.init) *
(.initcall3.init) *(.initcall3s.init) *(.initcall4.init) *(.initcall4s.init) *(.initcall5.init) *
(.initcall5s.init) *(.initcallrootfs.init) *(.initcall6.init) *(.initcall6s.init) *(.initcall7.init)
 *(.initcall7s.init) __initcall_end = .;
  __con_initcall_start = .; *(.con_initcall.init) __con_initcall_end = .;
  __security_initcall_start = .; *(.security_initcall.init) __security_initcall_end = .;
  . = ALIGN(4); __initramfs_start = .; *(.init.ramfs) . = ALIGN(8); *(.init.ramfs.info)
  __init_begin = _stext;
   *(.init.data) *(.cpuinit.data) *(.meminit.data) . = ALIGN(8); __ctors_start = .; *(.ctors)
__ctors_end = .; *(.init.rodata) . = ALIGN(8); __start_mcount_loc = .; *(__mcount_loc) __stop_
mcount_loc = .; *(.cpuinit.rodata) *(.meminit.rodata)
  }
  . = ALIGN((1 << 12)); .data..percpu : AT(ADDR(.data..percpu) - 0) { __per_cpu_load = .; __per_
cpu_start = .; *(.data..percpu..first) . = ALIGN((1 << 12)); *(.data..percpu..page_aligned) *
(.data..percpu..readmostly) *(.data..percpu) *(.data..percpu..shared_aligned) __per_cpu_end = .; }
  . = ALIGN((1 << 12));
  __init_end = .;
```

从这部分链接脚本可见 Linux 内核的起始位置是 0xC0008000，这个地址可以保证整个系统转入虚拟地址后，内核代码的地址映射仍然是一致的。0xC0008000 的位置是通过如下 vmlinux.lds.S 语句生成的。注意通常 CONFIG_XIP_KERNEL 都是不被配置的，另外宏 PAGE_

OFFSET 和 TEXT_OFFSET 都是和体系结构相关的。一般的 PAGE_OFFSET 是 0x80000000，而 TEXT_OFFSET 是 0x8000。

```
#ifdef CONFIG_XIP_KERNEL
    . = XIP_VIRT_ADDR(CONFIG_XIP_PHYS_ADDR);
#else
    . = PAGE_OFFSET + TEXT_OFFSET;
#endif
```

另外从生成的链接脚本可见，除了具体的处理器和体系结构信息（如 . proc. info. init）外，还有一些 init 的信息（主要是各种 init 的段），这些信息还是很重要的。之前讨论内核初始化需求的时候，提到初始化代码应该在初始化之后释放相应的内存空间，这里可以看出 Linux 内核已经将初始化相关的代码统一存放并管理，一方面方便初始化时调用，另一方面可以在系统初始化之后将这部分空间释放。这个文件还是很重要的，一个文件就解决了几个问题。

内核生成的 vmlinux 文件实在太大、不适合引导加载，所以需要进行压缩，Linux 内核压缩后为 zImage，可以说 zImage 是将可执行 Linux 内核 vmlinux 压缩后作为数据包含在自身中，图 4-6 展示了 vmlinux 到 zImage 的生成流程。

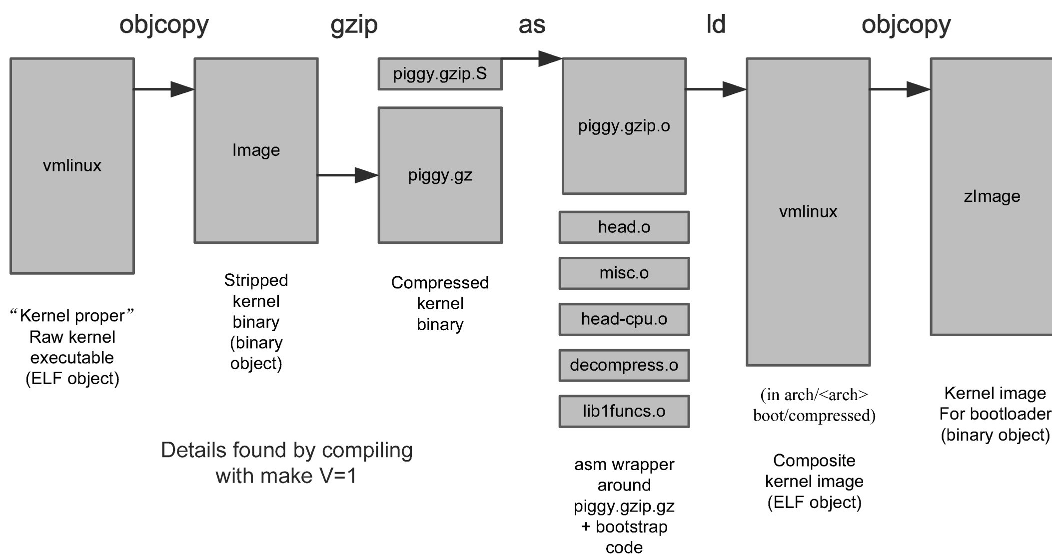


图 4-6　zImage 生成流程

zImage 的生成过程有一个问题，就是 zImage 究竟放在哪里，zImage 需要知道这个位置。因为它不能对这个位置进行任何的假设，并且需要该位置信息来判断解压后的空间是否会对自身有影响。另外对于具体的 memory 空间，地址信息是和具体处理器相关的，需要一个方式让与体系结构相关的 zImage 中包含这些信息，这样就可以自给自足了。下面看看 Linux 内核是如何实现的。

从图 4-6 中可见，压缩后的 vmlinux 会增加 arch/arm/boot/compressed 的代码作为头，然后生成 zImage，先来看看 zImage 头中 head. S 的代码，这里只是摘录了一部分：

```
start:
        .typestart,#function
  THUMB(      adrr12, BSYM(1f)      )
  THUMB(      bxr12      )
  THUMB(      .rept6      )
  ARM(      .rept8      )
        movr0, r0
        .endr

        b1f
        .word   0x016f2818   @ Magic numbers to help the loader
        .word   start        @ absolute load/run zImage address
        .word   _edata       @ zImage end address
1:      movr7, r1            @ save architecture ID
        movr8, r2            @ save atags pointer
......
#ifdef CONFIG_AUTO_ZRELADDR
        @ determine final kernel image address
        mov r4, pc
        and r4, r4, #0xf8000000
        add r4, r4, #TEXT_OFFSET
#else
        ldr r4, =zreladdr
#endif
        subs r0, r0, r1      @ calculate the delta offset
```

这部分代码会保存处理器架构 ID 信息和 atags 信息（用于传送启动参数等信息）。另外有个符号 zreladdr，该符号就是 zImage 最终被加载到内存中的地址。这个符号在哪里生成的呢？来看看 compressed 目录下的 Makefile 文件，其中有这么一段：

```
# Supply ZRELADDR to the decompressor via a linker symbol.
ifneq ($(CONFIG_AUTO_ZRELADDR),y)
LDFLAGS_vmlinux := --defsym zreladdr=$(ZRELADDR)
endif
```

这里直接通过链接工具创建符号 zreladdr，但数据是通过 ZRELADDR 参数传递的。这个参数又是如何来的呢？继续追踪 boot 目录下的 Makefile：

```
ifneq ($(MACHINE),)
include $(srctree)/$(MACHINE)/Makefile.boot
endif

# Note: the following conditions must always be true:
#   ZRELADDR == virt_to_phys(PAGE_OFFSET + TEXT_OFFSET)
#   PARAMS_PHYS must be within 4MB of ZRELADDR
```

```
#   INITRD_PHYS must be in RAM
ZRELADDR        := $(zreladdr-y)
PARAMS_PHYS := $(params_phys-y)
INITRD_PHYS    := $(initrd_phys-y)

export ZRELADDR INITRD_PHYS PARAMS_PHYS
```

这里可以看到 ZRELADDR 是由 zreladdr - y 定义的，又出来一个变量。为什么不直接定义呢？这样设计是为什么呢？之前提到内存地址在具体的处理器之间是有差别的，如何屏蔽这种差别？这种差别的配置还是要放入具体的处理器代码目录中。Linux 内核也是这个思路，这段代码的开始 include 一个文件 Makefile. boot，而且该文件是在一个具体的处理器目录中的，就是它进行了处理器差异的配置。具体到 DM 3730 等之前介绍的 TI 处理器都是引用自 mach - omap2 目录下的 Makefile. boot，其内容如下：

```
  zreladdr-y        :=0x80008000
params_phys-y       :=0x80000100
initrd_phys-y       :=0x80800000
```

查看一下芯片手册会发现，第一段内存的地址空间就是从 0x80000000 起始的地址，这样就对应上了。可见新的处理器只要在对应的目录中定义自己的 Makefile. boot（其中包括上面的定义）就能自动生成正确的 zImage。

到这里算是告一段落，但是嵌入式开发工程师通常比较熟悉 uImage，为什么不是 zImage 呢？这个和 u - boot 的 Boot loader 功能相关，主要是因为 zImage 知道自己所存放的地址，但是它还是需要 u - boot 将其放入合适的位置。u - boot 是通过 uImage 的开始一段的内容知道这个信息的，也就是说 uImage 实际是给 zImage 封装了一个简单的头，这个头中包括一些 u - boot 需要的信息。这就要求 Linux 内核能直接生成 uImage。来看看 uImage 是如何生成的？同样是在 arch/arm/boot 目录下的 Makefile：

```
quiet_cmd_uimage = UIMAGE   $@
      cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A arm -O linux -T kernel \
            -C none -a $(LOADADDR) -e $(STARTADDR) \
            -n' Linux - $(KERNELRELEASE)' -d $< $@

ifeq ($(CONFIG_ZBOOT_ROM),y)
 $(obj)/uImage: LOADADDR = $(CONFIG_ZBOOT_ROM_TEXT)
else
 $(obj)/uImage: LOADADDR = $(ZRELADDR)
endif

 $(obj)/uImage: STARTADDR = $(LOADADDR)

 $(obj)/uImage: $(obj)/zImage FORCE
     $(call if_changed,uimage)
    @echo'    Image  $@  is ready'
```

这里的 cmd_uimage 是 kbuild 用来定义客户自定义的编译命令，该命令的名字就是 uimage，实际调用 u-boot 的 mkimage 命令生成 uImage。在 DM 3730 等 TI 处理器中最终会将 ZRELADDR 即 0x80008000 作为参数传入，这样就正确地生成 uImage，完成了 Linux 内核 image 生成的全过程。

2. 处理器启动至 boot loader

先来看看芯片的启动时序，以 DM 3730 为例，上电时序如图 4-7 所示。

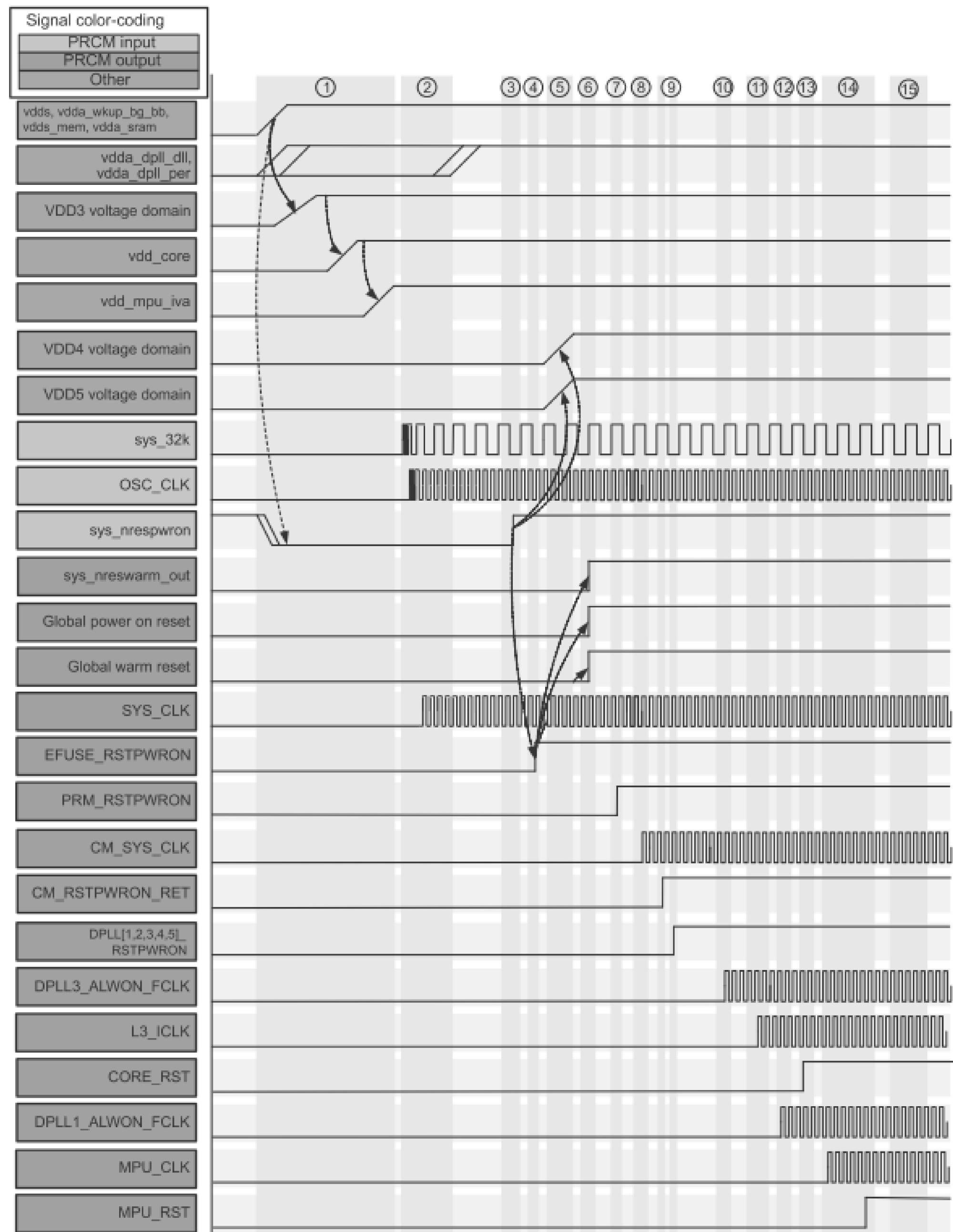


图 4-7　DM 3730 上电时序

图 4-7 引自《DM 3730 芯片手册》中第 266 页框图。从该图中可以看到整个芯片上电的硬件信号流程，这对了解整个系统的正常启动至关重要。另外可以看出系统的启动是从外部的供电和时钟信号开始，以主处理器的复位信号（MPU_RST）信号拉高为结束，之后就开始执行启动代码了。硬件启动时序正常的重要因素就是保证供电和时钟的稳定，要符合芯片的参数要求。

处理器上电启动代码的执行是和体系结构相关的，以 ARM 为核心的处理器，上电启动执行的第一个代码是 reset 向量的代码，reset 向量上电默认为 0x00000000 地址。指令执行需要 XIP 存储器，除了 ROM、RAM 之外，NOR Flash 也是 XIP 存储器，所以处理器是可以从 NOR 启动的。不同的处理器如何选择启动指令执行，比如转到 ROM Code 执行，就各有各的实现方式。下面是《DM 3730 芯片手册》中第 203 页的一段话：

The system has a 1 – MB boot space in the on – chip boot ROM or on the GPMC memory space. When booting from the on – chip ROM with the appropriate external sys_boot5 pin configuration, the 1–MB memory space is redirected to the on – chip boot ROM memory address space [0x4000 0000 – 0x400F FFFF]. When booting from the GPMC with the appropriate external sys_boot5 pin configuration, the memory space is part of the GPMC memory space. For more information about sys_boot5 pin configuration, see Chapter 10, Memory Subsystem, and Chapter 26, Initialization.

根据这段话可以明确 DM 3730 芯片设计时是如何解决该问题的。DM 3730 在系统启动时将 0x0 开始的 1MB 空间直接映射到 ROM Code 中，当启动时检查 sys_boot5 引脚来决定地址是否重定向，然后再 reset ARM 核心，这样使得 ARM 可以执行正确的代码来进行启动。

系统启动后，ARM 就开始执行 ROM Code 的代码。ROM Code 支持的启动设备称为启动能力，具体的启动能力是不同的处理器根据需求实现的。图 4-8 和图 4-9 分别是 DM 816X 和 DM 3730 的 ROM Code 框架。

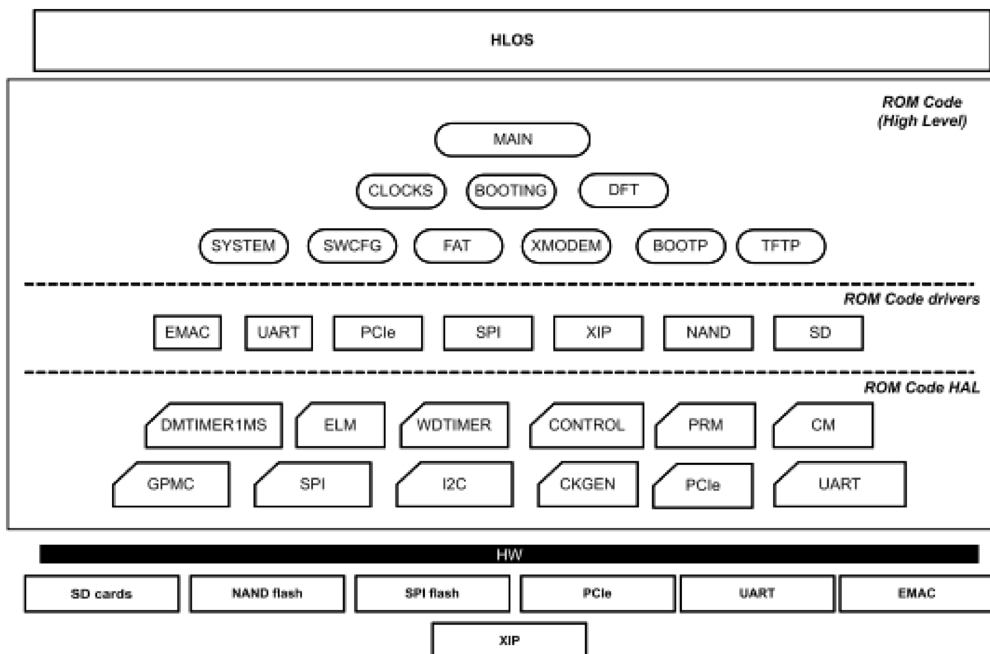


图 4-8　DM 816X 芯片 ROM Code 框架

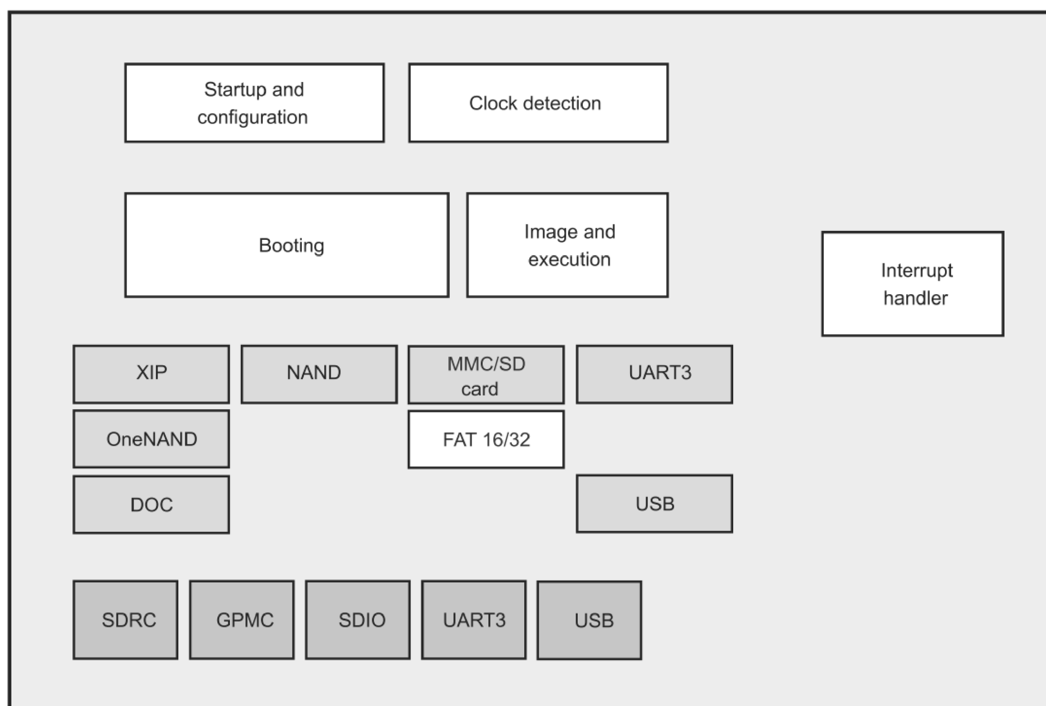


图 4-9 DM 3730 芯片 ROM Code 框架

从图 4-8 和图 4-9 中可见，DM 816X 的启动能力要比 DM 3730 强得多。DM 816X 支持网络启动和 PCIe 启动，这些都是 DM 3730 不具备的，也是符合 DM 816X 服务器以及 PCIe 板卡这些功能需求的。注意 DM 3730 中的 ROM Code 有 clock detection 模块，主要用来检测外部高速晶振的频率。通常 ROM Code 运行时，主处理器的时钟频率为外部高速晶振的频率（如 26 MHz）。这里 ROM Code 的检测模块确认时钟频率，可以为后续 boot loader 以及 Linux 内核设置各个不同模块的时钟提供基础时钟，进而明确如何进行配置才能得到需要的时钟频率。

ROM Code 的主要任务就是到其支持的设备中查找 boot loader，并加载到片内 RAM 中执行。这个过程中涉及两部分：

其一，识别正确的 boot loader 并加载，通常 ROM Code 会通过 image 增加特定的头标记，来识别并正确加载。芯片厂商会提供相应的生成和烧写工具代码以供使用。

其二，boot loader 编译的时候还是要以片内 RAM 的地址空间为基础，这就要在 boot loader 中进行设置。

先看看 DM 3730 是如何操作的。

x-loader 需要在片内内存中执行，其板级相关代码对应的目录中有 config. mk 文件，其中会设置代码的起始地址：

```
TEXT_BASE = 0x40208800
```

而在链接的时候会通过参数 -Ttext $(TEXT_BASE) 指定代码的起始地址为 TEXT_BASE 定义的地址。这样整个代码就是在片内内存空间。x-loader 加载到片内 RAM 后其内

存布局如图 4-10 所示。

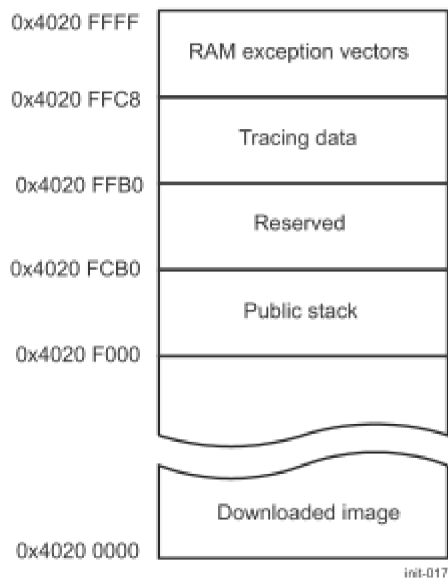


图 4-10　DM 3730 x-loader 执行片内 RAM 布局

x-loader 的最终 image 要加入特殊的头信息，具体说明如图 4-11 所示，图 4-11 引自《DM 3730 芯片手册》。其中包含加载的地址和 image 的大小信息。

Table 26-46. GP Device Software Image

Field	Non-XIP Device Offset	XIP Device Offset	Size (Bytes)	Description
Size	0x0000	–	4	Size of the image
Destination	0x0004	–	4	Address where to store the image
Image	0x0008	0x0000	x	Image

图 4-11　DM 3730 x-loader image 头信息说明

这个头信息是通过工具 signGP 生成的，相应信息如下：

```
x-load.bin.ift: signGP System.map x-load.bin
    TEXT_BASE = `grep -w _start System.map|cut -d' ' -f1'
    ./signGP x-load.bin $(TEXT_BASE)
    cp x-load.bin.ift MLO
```

又见到 TEXT_ BASE，这就要求加载的地址和编译的地址一一对应。

u-boot 则是在内存中执行，其地址应该是从 0x80000000 开始的空间。u-boot 中对应的 config.mk 的设置：

```
TEXT_BASE = 0x80e80000
```

可见其设置在内存的地址空间。这样代码所在空间的问题就解决了。但又出现一个问题，代码被加载到什么地址了呢？对于 x-loader，这个问题没办法回答，原因在于 ROM

Code 并不公开这些信息。u－boot 则是通过 x－loader 加载的，来看看相应的设置，在相应的板级配置文件中包含如下的信息：

```
#define CFG_LOADADDR                0x80008000
```

这个就是加载的地址，为什么和代码的地址不同呢？这个就无从考究了。这种不同会有问题吗？答案是不会的，主要是因为在跳转到加载地址执行代码（开始处为汇编代码），是使用的是相对地址即地址无关来执行，而一般跳转指令（如 b、bl 等）都是相对地址跳转，所以没有问题。但是如果 u－boot 中跳转到 start_armboot，该 C 函数使用的是 ldr pc 直接加载指令寄存器这种绝对地址跳转方式，此时代码的地址必须一一对应，否则这个跳转就会出现问题。至于为什么一定要进行这个绝对跳转，因为这是开发者设置的，只有开发者才最了解其内存布局。boot loader 会留出该接口给开发者设置，这样才能保证后续的内核加载等操作的正确进行。在绝对跳转之前，u－boot 和 x－loader 的 start. S 中都包含的搬移代码的重定向功能能保证代码地址和在地址空间中的位置一一对应。相应的重定向代码如下：

```
relocate:                                   /* relocate U-Boot to RAM */
    adr    r0, _start                       /* r0 <- current position of code */
    ldr    r1, _TEXT_BASE                   /* test if we run from flash or RAM */
    cmp    r0, r1                           /* don't reloc during debug */
    beq    stack_setup

    ldr    r2, _armboot_start
    ldr    r3, _bss_start
    sub    r2, r3, r2                       /* r2 <- size of armboot */
    add    r2, r0, r2                       /* r2 <- source end address */

copy_loop:                                  /* copy 32 bytes at a time */
    ldmia   r0!, {r3-r10}                   /* copy from source address [r0] */
    stmia   r1!, {r3-r10}                   /* copy to target address [r1] */
    cmp     r0, r2                          /* until source end addreee [r2] */
    ble    copy_loop
```

这些代码主要实现将代码移动到 TEXT_BASE 开始的地址，x－loader 也采用同样的办法解决加载地址和编译链接地址不匹配的问题。需要注意的是：这段代码还是比较简单的实现，其中并没有考虑一些特殊情况的覆盖问题（比如拷贝过程中的局部覆盖），所以加载代码的地址和编译链接的地址要有比较大的间隔，才能保证正确的结果。

再来看看 DM 816X 是如何操作的。

由于 DM 816X 片内 RAM 有 512 KB，完全可以加载并执行 u－boot，所以在该阶段只需要一次 boot loader。而 boot loader 同样是加载到片内 RAM 中执行。

DM 816X 由 ROM Code 加载的 image 的格式同 DM 3730，具体见《DM 816X 芯片手册》中第 2400 页的说明。在生成 image 方面，DM 816X 与 DM 3730 不同，DM 3730 使用的是单

独的 signGP 工具，而 DM 816X 使用的是 u - boot 中的 mkimage 工具。在 u - boot 中的 tools 目录下可以看到 tiimage 的 C 文件，这就是定义了 TI 特殊的 image 头及其生成方式，用于生成符合 DM 816X 处理器的 image 文件。生成代码是在 Makefile 文件中，具体内容如下：

```
$(obj)u-boot.ti:        $(obj)u-boot.bin
        $(obj)tools/mkimage -T tiimage \
        -e $(TI_LOAD_ADDR) -n $(TI_DEVICE) -d $< $(obj)$(TI_IMAGE)
```

该命令通过 mkimage 添加必要的头信息，如加载地址、设备信息，并生成 TI_IMAGE 定义名字的 image。

这里可见，除了 TI_IMAGE 之外,还有 TI_LOAD_ADDR 和 TI_DEVICE 两个变量需要明确。它们都是在哪里定义的呢？这些信息和板级信息相关，还记得之前提到的板级配置文件 config. mk 吧，就是在其中定义的，来看看具体内容：

```
# Output image name.  Used only in case of non-xip.  For NOR boot u-boot.bin
# could be used.
TI_IMAGE = u-boot.noxip.bin
# This will be used by mkimage extension to select header for image
TI_DEVICE = ti81xx
# ROM code will load u-boot to this address
TI_LOAD_ADDR = 0x40400000
TEXT_BASE = 0x80700000
CROSS_COMPILE := arm-none-linux-gnueabi-
LDSCRIPT := board/ti/ti8168/u-boot.lds
# Over-ride the macros if supplied from the Makefile
sinclude $(OBJTREE)/board/$(BOARDDIR)/config.tmp
```

这里定义了板级特殊的变量，其中 image 的加载地址是 0x40400000，即在片内内存的地址区域中。这里 TEXT_ BASE 是在 0x80700000，且为内存地址，这又如何操作呢？后续会进行说明。另外有一部分就是加载了特别的配置文件 config. tmp，其中的原因又是什么呢？主要是因为 DM 816X 支持从不同的设备启动，就要有不同的 u - boot 的 image，其中会有一些特殊的设置，这些特殊的设置可以通过 config. tmp 将这些变量重定义来实现。这样就开放一个重定义的接口文件来实现差异化。

具体的设置是在 Makefile 中通过不同的配置实现，具体内容如下：

```
ti8168_evm_config\
ti8168_evm_config_nand\
ti8168_evm_config_nor\
ti8168_evm_config_spi\
ti8168_evm_config_sd\
ti8168_evm_min_ocmc\
```

```
ti8168_evm_min_pcie_32\
ti8168_evm_min_pcie_64\
ti8168_evm_min_sd:unconfig
	@mkdir -p $(obj)include
	@echo "#define CONFIG_TI81XX"			>> $(obj)include/config.h
	@echo "#define CONFIG_TI816X"			>> $(obj)include/config.h
	@if [ "$(findstring _nand,$@)" ];then \
		echo "#define CONFIG_SYS_NO_FLASH"	>> $(obj)include/config.h ;\
		echo "#define CONFIG_NAND_ENV"		>> $(obj)include/config.h ;\
		echo "Setting up TI8168 NAND build with ENV in NAND..." ;\
	elif [ "$(findstring _nor,$@)" ];then \
		echo "#define CONFIG_NOR"		>> $(obj)include/config.h ;\
		echo "#define CONFIG_NOR_BOOT"		>> $(obj)include/config.h ;\
		echo "Setting up TI8168 NOR build with ENV in NOR..." ;\
	elif [ "$(findstring _spi,$@)" ];then \
		echo "#define CONFIG_SYS_NO_FLASH"	>> $(obj)include/config.h ;\
		echo "#define CONFIG_SPI_ENV"		>> $(obj)include/config.h ;\
		echo "#define CONFIG_TI81XX_SPI_BOOT"	>> $(obj)include/config.h ;\
		echo "Setting up TI8168 SPI build with ENV in SPI..." ;\
	elif [ "$(findstring _min_sd,$@)" ];then \
echo "#define CONFIG_SYS_NO_FLASH"			>> $(obj)include/config.h ;\
		echo "#define CONFIG_SD_BOOT"		>> $(obj)include/config.h ;\
		echo "TI_IMAGE = u-boot.min.sd"		>> $(obj)board/ti/ti8168/config.tmp;\
		echo "Setting up TI8168 SD boot minimal build..." ;\
	elif [ "$(findstring _min_pcie_32,$@)" ];then \
		echo "TEXT_BASE = 0x80700000"		>> $(obj)board/ti/ti8168/config.tmp;\
		echo "#define CONFIG_TI816X_MIN_CONFIG"	>> $(obj)include/config.h ;\
		echo "Setting up TI8168 minimal build for 1st stage..." ;\
		echo "#define CONFIG_SPI_BOOT"		>> $(obj)include/config.h;\
		echo "#define CONFIG_TI81XX_PCIE_BOOT"	>> $(obj)include/config.h ;\
		echo "#define CONFIG_SYS_NO_FLASH"	>> $(obj)include/config.h ;\
		echo "#define CONFIG_TI81XX_SPI_BOOT"	>> $(obj)include/config.h ;\
		echo "#define CONFIG_TI81XX_PCIE_32"	>> $(obj)include/config.h ;\
		echo "TI_IMAGE = u-boot.min"		>> $(obj)board/ti/ti8168/config.tmp;\
	elif [ "$(findstring _min_pcie_64,$@)" ];then \
		echo "TEXT_BASE = 0x80700000"		>> $(obj)board/ti/ti8168/config.tmp;\
		echo "#define CONFIG_TI816X_MIN_CONFIG"	>> $(obj)include/config.h ;\
		echo "Setting up TI8168 minimal build for 1st stage..." ;\
		echo "#define CONFIG_SPI_BOOT"		>> $(obj)include/config.h;\
		echo "#define CONFIG_TI81XX_PCIE_BOOT"	>> $(obj)include/config.h ;\
```

```
        echo "#define CONFIG_SYS_NO_FLASH"        >> $(obj)include/config.h ;\
        echo "#define CONFIG_TI81XX_SPI_BOOT"     >> $(obj)include/config.h ;\
        echo "#define CONFIG_TI81XX_PCIE_64"      >> $(obj)include/config.h ;\
        echo "TI_IMAGE = u-boot.min"              >> $(obj)board/ti/ti8168/config.tmp;\
    elif [ "$(findstring _sd,$@)" ] ;then \
        echo "#define CONFIG_SYS_NO_FLASH"        >> $(obj)include/config.h ;\
        echo "#define CONFIG_MMC_ENV"             >> $(obj)include/config.h ;\
        echo "Setting up TI8168 SD build with ENV in MMC..." ;\
    elif [ "$(findstring _ocmc,$@)" ] ;then \
        echo "#define CONFIG_SYS_NO_FLASH"        >> $(obj)include/config.h ;\
        echo "#define CONFIG_MINIMAL"             >> $(obj)include/config.h ;\
        echo "TEXT_BASE = 0x40410000"             >> $(obj)board/ti/ti8168/config.tmp;\
        echo "Setting up TI8168 minimal build..." ;\
    else\
        echo "#define CONFIG_SYS_NO_FLASH"        >> $(obj)include/config.h ;\
        echo "#define CONFIG_NAND_ENV"            >> $(obj)include/config.h ;\
        echo "Setting up TI8168 default build with NAND..." ;\
    fi;
    @ $(MKCONFIG) -a ti8168_evm arm arm_cortexa8 ti8168 ti ti81xx
```

从中可见，针对不同启动设备的 image，会有两个配置文件需要进行相应的设置。一个是include/config.h，另一个就是板级的 config.tmp。通过这两个文件，就可以生成各种不同启动设备的 image 了。

从整个过程可见，不同的处理器的启动思路基本是相同的，相应的差别主要在设备地址以及不同设备的启动能力上。

通常在该部分并没有信息输出到控制台，一般的输出都是要在 boot loader 阶段才开始的。

3. boot loader 至内核 image

在 ROM Code 将 boot loader 加载到片内内存之后，就会跳转到相应的地址执行 boot loader 的代码。之前已经提到，boot loader 被加载的地址和编译的地址可能不同，需要有个重定向的过程。而重定向需要从片内 RAM 重定向到内存的空间，而这时内存控制器等并没有进行初始化，要求在重定向之前对内存控制器进行初始化。这样就会将整个 boot loader 的流程实际分割成两个阶段，即重定向之前的阶段和重定向之后的阶段。相应的分割点就是之前提到的 start.S 中的实际地址跳转指令，内容如下：

```
    ldrpc, _start_armboot/* jump to C code          */

_start_armboot:.word start_armboot
```

DM 3730 的 x-loader 其实是基于 u-boot 进行的精简，所以框架和 u-boot 是一致

的。后面都是以 u-boot 为例进行说明。

第一阶段的主要工作流程如图 4-12 所示。

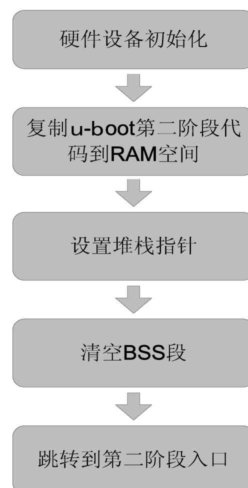


图 4-12　boot loader 第一阶段流程

首先，注意第一阶段由于其所在的空间并不是实际编译的空间，所以跳转指令必须使用相对跳转。在这一阶段主要初始化的硬件就是内存。对于u-boot 框架来说，如果处理器有特别的底层初始化操作需要进行，通常会实现 lowlevel_init。这一阶段的最终内存初始化操作一般都是通过定义 s_init 函数实现的。在这一阶段都会有汇编指令相对跳转到 C 函数 s_init 中执行。分别看看 DM 3730 和 DM 816X 中相应函数都进行了哪些操作。

DM 3730 的 x-loader 相应代码如下：

```
void s_init(void)
{
    watchdog_init();
#ifdef CONFIG_3430_AS_3410
    /* setup the scalability control register for
     * 3430 to work in 3410 mode
     */
    __raw_writel(0x5ABF,CONTROL_SCALABLE_OMAP_OCP);
#endif
    try_unlock_memory();
    set_muxconf_regs();
    delay(100);
    prcm_init();
    per_clocks_enable();
    config_3430sdram_ddr();
}
```

DM 816X 的 u-boot 相应的代码如下：

```
void s_init(u32 in_ddr)
{
    /*
     * Disable Write Allocate on miss to avoid starvation of other masters
     * (than A8).
     *
     * Ref DM 816x Erratum: TODO
     */
    l2_disable_wa();
    l2_cache_enable();        /* Can be removed as A8 comes up with L2 enabled */
#ifdef CONFIG_SETUP_1V
```

```
        __raw_writel(0x102, 0x4818155c);
        while((__raw_readl(0x4818155c) & 0x3)! =0x2);

        __raw_writel(0x102, 0x48181560);
        while((__raw_readl(0x48181560) & 0x3)! =0x2);

        __raw_writel(0x00000001, 0x4803213c);
        __raw_writel(0xfffffff0, 0x48032134);
#endif

        prcm_init(in_ddr);              /* Setup the PLLs and the clocks for the peripherals */
        set_muxconf_regs();
        if (! in_ddr)
            config_ti816x_sdram_ddr();  /* Do DDR settings */
#ifdef CONFIG_TI816X_VOLT_SCALE
        voltage_scale_init();
#endif
}
```

从中可见，这部分初始化主要进行引脚设置、电源和时钟设置，以及内存控制器的设置。当然不同处理器的内存控制器设置是不同的，通常芯片厂商会提供相应的工具，如图 4-13 所示，根据内存手册的参数得到对应寄存器的值，在相应的函数中对控制寄存器写

	OMAP35x register name	OMAP35x register bit length	Memory datasheet symbol	Memory Datasheet value	unit	OMAP35x Setting (Decimal)
	tCK		tCK	5	ns	
SDRC_ACTIM_CTRLA_p	TRFC	5	tRFC	72	ns	15
	TRC	5	tRC	55	ns	11
	TRAS	4	tRAS	40	ns	8
	TRP	3	tRP	15	ns	3
	TRCD	3	tRCD	15	ns	3
	TRRD	3	tRRD	10	ns	2
	TDPL(tWR)	3	tWR	15	ns	3
	TDAL	5	tDAL	6	tCK	6
SDRC_ACTIM_CTRLB_p	TWTR	2	tWTR	2	tCK	2
	TCKE	3	tCKE	1	tCK	1
	TXP	3	tXP	2	tCK	2
	TXSR	8	tXSR	112.5	ns	23
SDRC_RFR_CTRL_p	ARCV	16	tREFI	7.8	us	1510

SDRC_ACTIM_CTRLA_p								
TRFC[31:27]	TRC[26:22]	TRAS[21:18]	TRP[17:15]	TRCD[14:12]	TRRD[11:9]	TDPL[8:6]	reserved	TDAL [4:0]
F	B	8	3	3	2	3	0	6
01111	01011	1000	011	011	010	011	0	00110
Register value (hex)	7AE1B4C6							

SDRC_ACTIM_CTRLB_p						
reserved[31:18]	TWTR[17:16]	reserved[15]	TCKE[14:12]	reserved[11]	TXP[10:8]	TXSR[7:0]
0	2	0	1	0	2	17
00000000000000	10	0	001	0	010	00010111
Register value (hex)	00021217					

SDRC_RFR_CTRL_p			
reserved[31:24]	ARCV[23:8]	reserved[7:2]	ARE[2:0]
00	05E6	0	1
Register value (hex)	0005E601		

图 4-13　DM 3730 内存控制器寄存器计算工具

入即可。上述的操作执行完毕，外部内存就可以访问了。

内存可以访问之后，就能够进行代码搬移重定向操作，进入第二阶段。

第二阶段的基本流程如图 4-14 所示。

第二阶段主要进行外部设备初始化操作，这些设备可以进行 kernel 以及初始文件系统加载，所以究竟初始化什么设备是可以定制的。之前介绍的配置文件，就可以通过宏来进行这部分定制。相应的设备初始化后，boot loader 就进入主循环，等待命令执行，或者直接执行设置好的命令。相应的命令主要用于通过设备加载内核 image 文件。不要忘记 boot loader 有一个重要的任务就是为内核传递参数，这就要求可以在 boot loader 中设置参数，boot loader 主要通过其 bootargs 为内核传递启动参数。u - boot 中具体的命令和参数在此就不做详细地介绍了。

第二阶段最终会通过 bootm 命令跳转到内核执行。在跳转进内核 image 执行之前，还要进行一些必要的准备操作，主要如下：

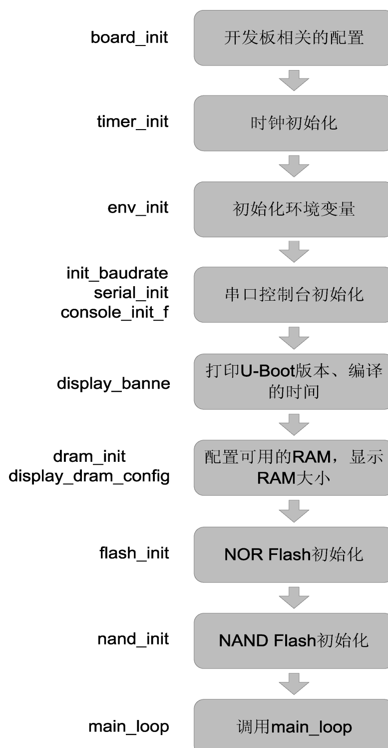


图 4-14　boot loader 第二阶段流程

```
char  * commandline = getenv ("bootargs")
s = getenv ("machid")
setup_start_tag (bd)
setup_end_tag (bd)
cleanup_before_linux()
    disable_interrupts()
    icache_disable()
    dcache_disable()
    cache_flush()
theKernel (0, machid, bd -> bi_boot_params);
```

这些准备操作是准备参数、关中断、清理 cache 等，最终通过 theKernel 这一指针跳转到内核 image 执行。为什么一定要进行这些操作呢？这些操作的前世今生又是怎样的呢？其实这些操作都是 Linux 内核规定 boot loader 在进入内核之前必须做的事情，毕竟 boot loader 和 Linux 内核之间统一的接口是有规定的。而 Linux 内核需要对处理器进行很多操作，这些操作是在特权级别并且对系统初始化很重要，自然会对处理器所处的状态等有严格的规定。关于 ARM 体系结构处理器的相关限制在内核的 Documentation/arm/Booting 文件中进行了详细的说明。从如下内容中可见对上述操作的要求。

```
In any case, the following conditions must be met:
- Quiesce all DMA capable devices so that memory does not get corrupted by bogus network packets or disk data. This will save you many hours of debug.
```

– CPU register settings

r0 = 0,

r1 = machine type number discovered in (3) above.

r2 = physical address of tagged list in system RAM, or physical address of device tree block (dtb) in system RAM

– CPU mode

All forms of interrupts must be disabled (IRQs and FIQs)

For CPUs which do not include the ARM virtualization extensions, the CPU must be in SVC mode. (A special exception exists for Angel)

CPUs which include support for the virtualization extensions can be entered in HYP mode in order to enable the kernel to make full use of these extensions. This is the recommended boot method for such CPUs, unless the virtualisations are already in use by a pre – installed hypervisor.

If the kernel is not entered in HYP mode for any reason, it must be entered in SVC mode.

– Caches, MMUs

The MMU must be off.

Instruction cache may be on or off.

Data cache must be off.

If the kernel is entered in HYP mode, the above requirements apply to the HYP mode configuration in addition to the ordinary PL1 (privileged kernel modes) configuration. In addition, all traps into the hypervisor must be disabled, and PL1 access must be granted for all peripherals and CPU resources for which this is architecturally possible. Except for entering in HYP mode, the system configuration should be such that a kernel which does not include support for the virtualization extensions can boot correctly without extra help.

– The boot loader is expected to call the kernel image by jumping directly to the first instruction of the kernel image.

On CPUs supporting the ARM instruction set, the entry must be made in ARM state, even for a Thumb –2 kernel.

以上部分摘自 Linux 内核中的文档，其中对新的技术（如虚拟化和设备树）进行了详细规定。这些规定在做实际的内核移植是需要注意的。

除了对处理器状态等的规定之外，有一点比较重要的就是 boot loader 和 Linux 内核之间如何进行参数的交互。这部分主要是通过 tagged list 来完成的。对于 tagged list 来说，也是由 Linux 内核进行规定的，同样是在 Booting 文件中有相关的描述。具体的解释如下（这里直接进行了翻译）：

boot loader 必须创建和初始化内核的 tagged list。一个合法的 tagged list 开始于 ATAG_CORE 并结束于 ATAG_NONE。ATAG_CORE tag 可以为空。一个空的 ATAG_CORE tag 的 size 字段设为“2”（0x00000002）。ATAG_NONE 的 size 字段必须设为“0”。tagged list 可以有任意多的 tag。boot loader 必须至少传递系统内存的大小和位置，以及根文件系统的位置，一个最小化的 tagged list

应该如下：

```
                 +----------+
base ->          | ATAG_CORE |  |
                 +----------+  |
                 | ATAG_MEM  |  | increasing address
                 +----------+  |
                 | ATAG_NONE |  |
                 +----------+  v
```

tagged list 应该放在内核解压后和 initrd 的“bootp”程序都不会覆盖的内存区域。ARM 体系结构下，建议放在 RAM 的起始的 16 KB 大小的地方。

通过这段说明，回答了之前并没有解答的疑问 zImage 的起始地址设置为 0x80008000，并没有从内存的首地址开始，而是从 32 KB 开始，这样为 boot loader 和参数的传递留下足够的空间，另外后面也会看到内核在这 32 KB 空间中还有需要保存的数据。

具体的参数及其传递的内容可以通过 u-boot 中 include/asm-arm/setup.h 文件获得，主要的标签信息如下：

```
struct tag {
    struct tag_header hdr;
    union {
        struct tag_corecore;
        struct tag_mem32mem;
        struct tag_videotextvideotext;
        struct tag_ramdiskramdisk;
        struct tag_initrdinitrd;
        struct tag_serialnrserialnr;
        struct tag_revisionrevision;
        struct tag_videolfbvideolfb;
        struct tag_cmdlinecmdline;

        /*
         * Acorn specific
         */
        struct tag_acornacorn;

        /*
         * DC21285 specific
         */
        struct tag_memclkmemclk;
    } u;
};
```

从中可见，为扩展留出一定的余地，只是要进行参数的扩展还是需要在 Linux 内核侧做特别的定制工作。

通常该阶段结束会输出如下信息：

```
## Booting kernel from Legacy Image at 81000000 ...
   Image Name:   Linux -2.6.37_DM 8127_IPNC_3.80.00
   Created:      2013-11-27   6:47:24 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2493848 Bytes =2.4 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...
```

接下来就进入 Linux 内核了。

4. 内核 image 至 start_kernel

在介绍 Linux 内核 image 生成中，已经介绍了压缩 image 的生成。所以要进入 Linux 内核 image，首先要做的就是将压缩的 image 进行解压缩操作。这部分是从 boot/compressed 目录下 head.S 文件中 start 处开始的，详细的流程如图 4-15 所示。

解压缩的过程控制台会输出如下信息：

```
Uncompressing Linux... done, booting the kernel.
```

解压缩结束后，最终通过__enter_kernel 加载 PC 指针进入真正的内核，进行后续的操作。

这个入口在 arch/arm/kernel/head.S 中，该文件就是 Linux 内核真正启动的地方，是初始化部分的开始，用汇编写成，为后面的 C 代码初始化做好准备。其中有一些宏定义需要明确它们的含义，具体见表 4-1。

表 4-1　重要的宏定义

宏	代码位置	默认值	说明
KERNEL_RAM_ADDR	arch/arm/kernel/head.S	0xC0008000	内核在内存中的虚拟地址
PAGE_OFFSET	arch/arm/include/asm/memory.h	0xC0000000	内核虚拟地址空间的起始地址，通常是内核配置 CONFIG_PAGE_OFFSET 设置的
TEXT_OFFSET	arch/arm/Makefile	0x00008000	内核起始位置相对于内存起始位置的偏移，一般由 textofs -y：=0x00008000 定义
PHYS_OFFSET	arch/arm/include/asm/memory.h	处理器相关	物理内存的起始地址，DM 3730 芯片是在 mach - omap2/include/mach/memory.h 包含的头文件 plat - omap/include/mach/memory.h 中定义

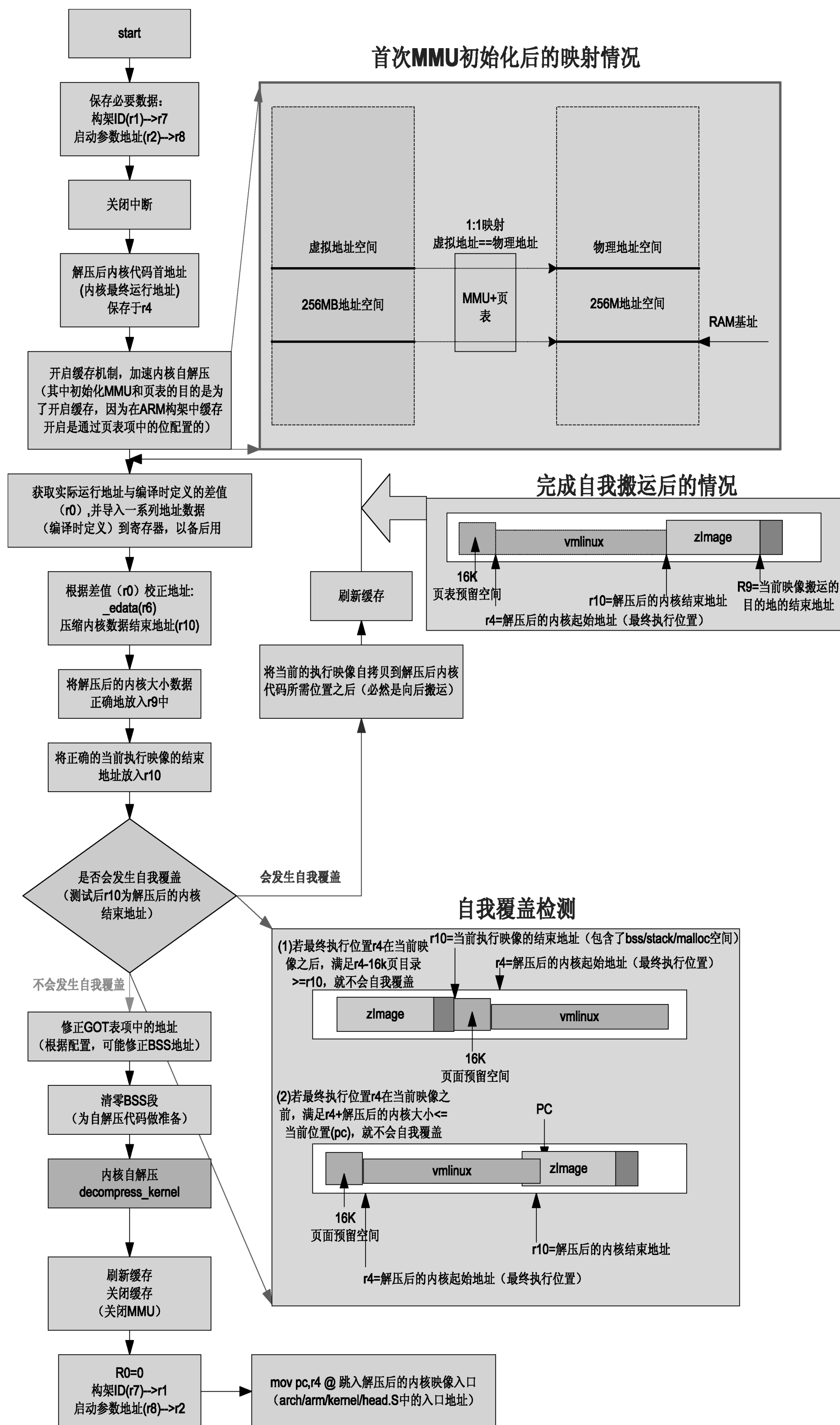


图 4-15　Linux 内核解压缩流程

这些宏定义之前也见过一些，这里进行详细说明也凸显出它们的重要性。这个阶段和移植相关的操作，主要就是这些宏定义的设置了。

注意这里主要是进行体系结构相关的初始化操作，主要的操作如下：

首先，检查处理器类型，通过__lookup_processor_type 实现（arch/arm/kernel/head-common.S 中），具体读取 processor ID 寄存器，并查找 image 中支持的体系结构列表；然后，通过__lookup_machine_type（arch/arm/kernel/head-common.S 中）查找 image 中支持的处理器架构列表；最后，初始化页表，该页表在 16 KB～32 KB 之间（ARM 体系结构下主页表的大小就是 16 KB，这样内存的前 32 KB 就都有用处了）通过 __create_page_tables 实现，该页表足够内核进行后续初始化操作。

处理器特别的操作通常是__switch_data 和 __enable_mmu（arch/arm/mm/proc-*.S 中实现）。后续还有初始化 TLB、caches，最终使能 MMU。紧接着就跳转到虚拟地址执行。下面来看看如何实现这个跳转的。注意在跳转到虚拟地址之前通常还是用相对跳转进行函数调用的。

```
            ldr r13, =__mmap_switched       @ address to jump to after mmu has been enabled
                                            @ 这里绝对地址是虚拟地址
            adr lr, BSYM(1f)                @ return (PIC) address
        ARM(add pc, r10, #PROCINFO_INITFUNC)
        THUMB(add r12, r10, #PROCINFO_INITFUNC)
        THUMB(mov pc, r12)
    1:  b__enable_mmu
    ….
        __turn_mmu_on:
            mov r0, r0
            mcr p15, 0, r0, c1, c0, 0       @ write control reg
            mrc p15, 0, r3, c0, c0, 0       @ read id reg
            mov r3, r3
            mov r3, r13                     @这里将之前保存的虚拟地址取出
            mov pc, r3                      @调转到相应的虚拟地址
```

这一阶段的具体流程如图 4-16 所示。

Linux 内核到这里，就要进入 start_kernel 来进行 C 代码级别的初始化操作了。DM 3730 的流程中调用和地址关系如图 4-17 所示。

5. start_kernel 至 rest_init

start_kernel 函数是在 Linux 内核的 init 目录下的 main.c 文件中。看到文件名就知道，这其中包含了 Linux 内核初始化的主要函数，而 start_kernel 则是重中之重，如此重要的函数有必要完整读一遍。这里以 2.6.37 内核对应函数为例，并会将解释说明和代码混合在一起（直接在每行代码之前以注释的形式），这样可以结合代码一起理解。该函数也是了解 Linux 内核各个模块一个非常好的入口。

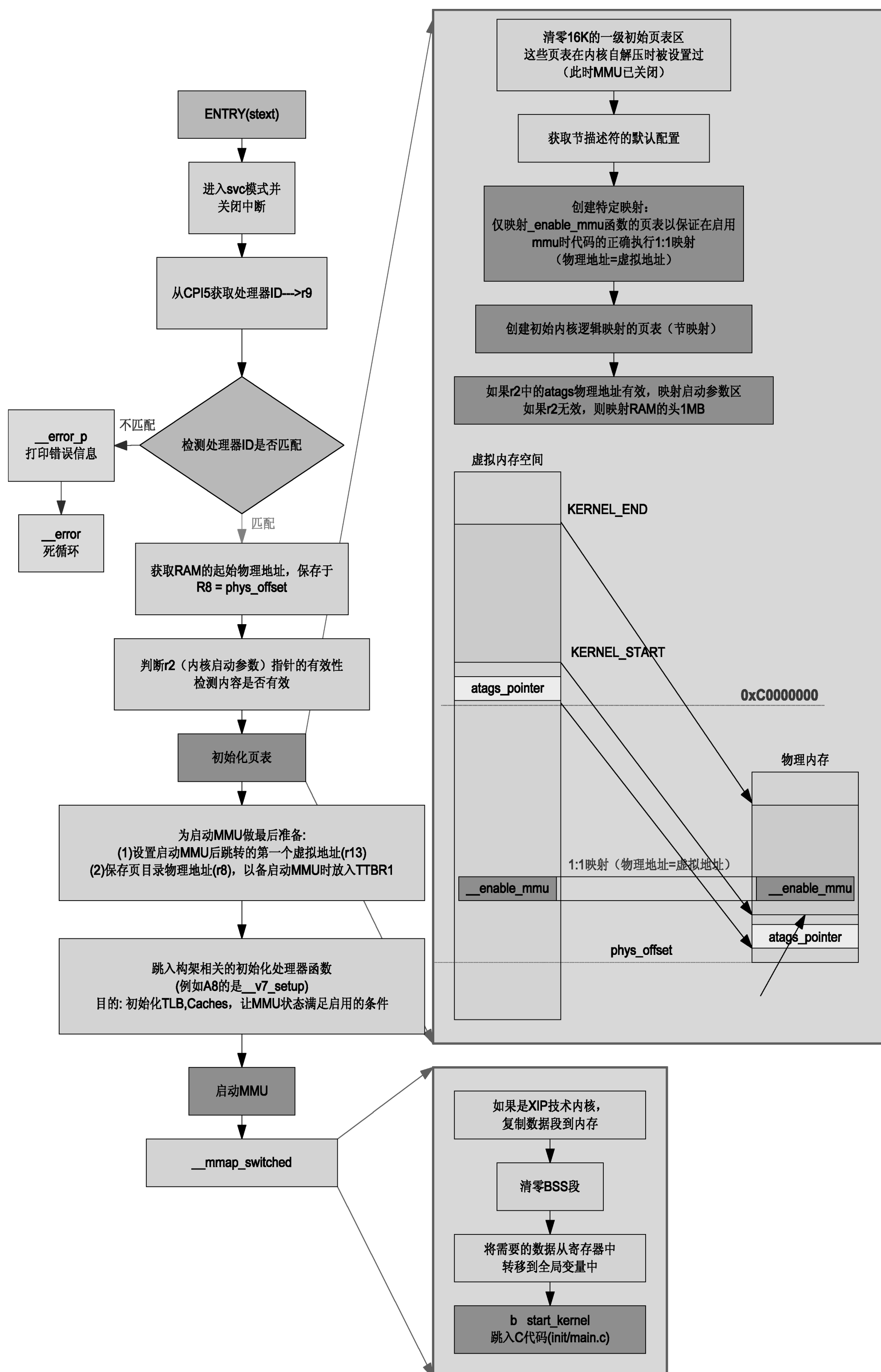


图 4-16 Linux 内核进入 C 代码之前的初始化流程

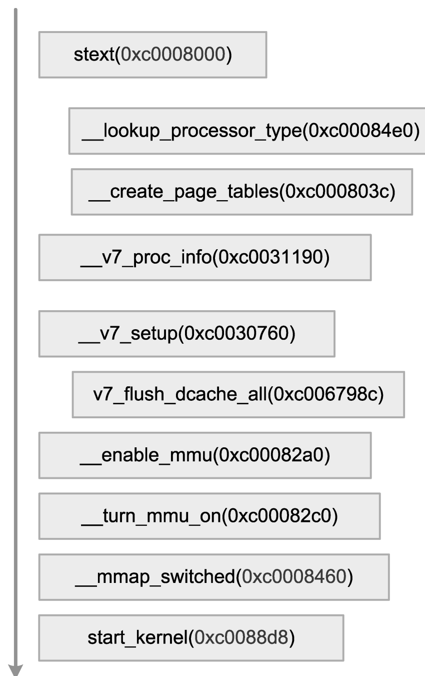


图 4-17　DM 3730 初始化流程调用和地址关系

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    //这里的变量是在链接时产生的,Linux 内核对参数的管理是将它们统一放到一个 section
    //_param 中,这样就需要知道 setction 的起始和结束地址,在链接脚本中通过 include/asm -
    //generic/vmlinux.lds.h 将这两个变量加入
    extern const struct kernel_param __start___param[], __stop___param[];
    //这里试图对 SMP 架构的不同处理器进行逻辑标定,具体的处理器号通常是硬件做的标记,
    //所以这里通常是启动时启动处理器号的逻辑映射。注意该函数在 main.c 中由属性 weak
    //定义,这样体系结构相关代码可以重定义该函数。对于老的内核 ARM 体系结构,一般这
    //是空函数;对于新的内核,该函数进行处理器逻辑标定。处理器逻辑标定功能对于最新
    //的 ARM 系统结构中大核小核(big.little)也是需要的
    smp_setup_processor_id();

    /*
     * Need to run as early as possible, to initialize the
     * lockdep hash:
     */
    //这里是内核调试相关功能的初始化
    //lockdep 是用来检查内核中各种锁的状态,用于检查、调试内核因不正确的使用锁造成的
    //死锁问题,在内核 debug 配置中有相应的功能配置
    lockdep_init();
```

```
    //内核中很多动态对象是有生命周期的,debug object 作为库来提供给其他模块,用于调试模
    //块内(如 timer)对动态对象的非法操作问题(如有效对象非法释放)
    debug_objects_early_init();

    /*
     * Set up the the initial canary ASAP:
     */
    //stack canary 用于检测栈溢出,canary 是金丝雀,古时候使用金丝雀来检查煤矿中是否安全,
    //这里就是使用这个典故。通常的做法是在栈上做一个特殊值作为标记,这个值需要在这里
    //进行初始化,该标记值就应该叫金丝雀
    boot_init_stack_canary();

    //cgroup 是 control group 的缩写,是 Linux 内核提供的一种可以限制、记录、隔离进程组(process
    //groups)所使用的物理资源(如 CPU、memory、IO 等)的机制。cgroup 是一种管理资源的方
    //式,也是后来才引进 Linux 内核的。这里是该功能基本管理数据的初始化操作,该功能还
    //在很多云应用中使用
    cgroup_init_early();

    //在内核没有准备好接收中断之前,先将中断关闭。中断通常是随机事件,发生问题时的
    //现象也是随机的,所以在内核准备好接收中断之前要主动关闭中断
    local_irq_disable();
    //这里是对启动时中断关闭进行状态标定,该标定主要用于特别操作的状态检查,如果检查
    //到与状态有关联操作的执行则可以进行报警
    early_boot_irqs_off();

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    //这里是 Linux 内核时间系统的部分功能的初始化。tick 是滴答,滴答和时间的间隔相关,而
    //tick 在 Linux 内核中是驱动管理时间的框架,这里的初始化主要是将 tick 提供的控制管理
    //接口注册到系统中
    tick_init();
    //标记当前启动的 CPU 的状态为激活状态
    boot_cpu_init();
    //这里是针对 highmem 的管理实体的初始化。highmem 的每个 page 需要进行单独映射,所以
    //这里初始化相应的地址映射管理结构
    page_address_init();
    //输出 Linux version 2.6.37 …等编译内核相关的信息
    printk(KERN_NOTICE "%s", linux_banner);
    //体系结构相关的初始化,并且获得内核的启动参数。其中会涉及中断向量表的重定位以及
    //板级初始化信息的加载。这个重要的初始化步骤后续会详细说明。体系结构相关初始化
    //过程中会输出一些处理器以及板级的信息
    setup_arch(&command_line);
```

```
    //这里指定 Linux 内核的内存管理结构的 owner 为内核初始的 init_task 进程,该功能通过
    //CONFIG_MM_OWNER 进行配置,通常不配置该功能
    mm_init_owner(&init_mm, &init_task);
    //将启动参数进行保存以便后续处理
    setup_command_line(command_line);
    //确认支持多少处理器
    setup_nr_cpu_ids();
    //初始化每个 CPU 的私有空间,这些私有空间存放 CPU 的私有数据。这部分数据可以不使
    //用自旋锁进行保护,这样可以提高性能。单处理器和 SMP 处理器不同
    setup_per_cpu_areas();
    //这里属于标记启动 CPU 的根进程
    smp_prepare_boot_cpu();/* arch-specific boot-cpu hooks */

    //下面是对于物理页管理的相关初始化
    //内存物理页分为不同的 zone 区域,对所有的 node 中的 zone 管理进行初始化
    build_all_zonelists(NULL);
    //对所有的处理器建立 page 分配通知机制
    page_alloc_init();

    //这里把 boot loader 传入的命令行进行输出
    printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
    //进行早期的参数解析,主要是和控制台相关
    parse_early_param();
    //进行完整的参数解析,static_command_line 是之前 static_command_line 保存的
    parse_args("Booting kernel", static_command_line, __start___param,
           __stop___param - __start___param,
           &unknown_bootoption);
/*
  * These use large bootmem allocations and must precede
  * kmem_cache_init()
  */
    //初始化 O(1)调度算法需要的 PID 哈希表
    //这里会输出信息:PID hash table entries: 512 (order: -1, 2048 bytes)
    pidhash_init();
    //VFS 早期的初始化,主要还是管理哈希表的初始化
    //这里会输出信息
    //Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
    //Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
    vfs_caches_init_early();
    //对内核特定的异常表( exception table )初始化,按照异常号大小进行排序
    //这里是内核提供的一些异常修复功能,散布在内核很多地方,属于横切功能
    sort_main_extable();
```

```
//体系结构相关的 trap 初始化。ARM 体系结构下为空,内核中断异常使用向量表的初始化
//在 early_trap_init 中进行
trap_init( );
//现在就可以对内核标准的内存分配方式进行初始化。该初始化之后,各个模块就可以
//使用标准的内核内存分配方式分配和使用内存。在此之前的初始化,工作内核提供了
//bootmem 分配方式,这里一并释放相应的资源
//注意这里只是初始化分配器,对于实际物理页资源的初始化是在和体系结构相关的 setup_
//arch 中进行的。这里会输出物理内存信息和虚拟地址映射信息
mm_init( );
/ *
  * Set up the scheduler prior starting any interrupts (such as the
  * timer interrupt). Full topology setup happens at smp_init( )
  * time - but meanwhile we still have a functioning scheduler.
  */
//初始化调度器的管理结构,主要是各个 CPU 的各种优先级管理资源以及运行队列。注意
//idle作为一个特殊的任务,需要在每个处理器上进行特别的标注,以便调度时进行特别操作
sched_init( );
/ *
  * Disable preemption - early bootup scheduling is extremely
  * fragile until we cpu_idle( ) for the first time.
  */
//虽然调度器已经准备就绪,但是不应该在系统初始化期间将初始化的任务抢占,所以这里
//禁止内核抢占。等到后续系统初始化完成,再打开该功能
preempt_disable( );
//内核抢占功能关闭,但是担心之前体系结构相关初始化打开中断,导致后续的初始化出现问
//题,所以这里检查一下是否中断 disable,这样可以提早发现,提早报警,以避免一些随机的问题
if (! irqs_disabled( )) {
    printk(KERN_WARNING "start_kernel( ): bug: interrupts were "
        "enabled *very* early, fixing it\n");
    local_irq_disable( );
}
//内核中 Read - Copy - Update 的初始化,该功能主要是在 SMP 下一种高效的读写锁机制,
//通过限定更新的时间点来进行数据保护和同步,可以配置不同的实现方式
rcu_init( );
//初始化 Radix 树,是一种快速查找的搜索树。在 Linux 内核中,包括 cache 机制等多种功
//能都采用该数据结构加速
radix_tree_init( );
/ * init some links before init_ISA_irqs( ) */
//初始化 Linux 内核的中断处理模块,主要是体系结构无关的中断描述结构
early_irq_init( );
//这里不仅是体系结构相关还是处理器相关的板级中断初始化操作,相应的函数接口是在
//setup_arch 中加载的
```

```
init_IRQ();
//初始化基于 radix 的优先级查找树,该数据结构主要用在虚拟地址空间的管理中
prio_tree_init();
//这里进行 timer 管理结构的初始化。如果是 SMP,会初始化每个 CPU 的 timer 管理结构
init_timers();
//初始化高精度定时器功能。主要提高系统的时钟精度(到纳秒级别)
hrtimers_init();
//初始化软中断等中断处理后半部的功能
softirq_init();
//初始化 Linux 内核的时间更新系统,该系统功能是保证系统时间的正确和稳定
timekeeping_init();
//对于系统相关的系统时钟进行初始化,系统时钟指定是在 setup_arch 中,从板级信息中
//获得的
time_init();
//初始化 Linux 内核的性能调试功能
profile_init();
//之前执行过体系结构相关的代码,为了避免问题,这里再检查一遍中断是否被打开
if (!irqs_disabled())
	printk(KERN_CRIT "start_kernel(): bug: interrupts were "
		"enabled early\n");
//之前标记系统状态,检查中断关闭状态,现在可以设置为中断打开了
early_boot_irqs_on();
//进行中断使能操作
local_irq_enable();

/* Interrupts are enabled now so all GFP allocations are safe. */
//这里可以允许所有情况下的物理内存页分配功能
gfp_allowed_mask = __GFP_BITS_MASK;

//内核内存中 cache 分配器的后续初始化工作。有 slab、slob 和 slub 几种实现
kmem_cache_init_late();

/*
 * HACK ALERT! This is early. We're enabling the console before
 * we've done PCI setups etc, and console_init() must be aware of
 * this. But we do want output early, in case something goes wrong.
 */
//控制台初始化,这样可以尽早看到启动信息
console_init();
//之前如果解析参数有问题,这里可以输出相应信息
if (panic_later)
	panic(panic_later, panic_param);
```

```
	//输出 lockdep 功能的属性及使用资源的信息
	lockdep_info();

	/*
	 * Need to run this when irqs are enabled, because it wants
	 * to self-test [hard/soft]-irqs on/off lock inversion bugs
	 * too:
	 */
	//在系统运行之前检查一下各种锁的功能是否正常,主要用于调试
	locking_selftest();

#ifdef CONFIG_BLK_DEV_INITRD
	//对 initrd 进行位置检查
	if (initrd_start && !initrd_below_start_ok &&
		page_to_pfn(virt_to_page((void *)initrd_start)) < min_low_pfn) {
		printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
			"disabling it.\n",
			page_to_pfn(virt_to_page((void *)initrd_start)),
			min_low_pfn);
		initrd_start = 0;
	}
#endif
	//这里是 memory cgroup 功能初始化。memory cgroup 是 cgroup 体系中提供的对于 memory 资
	//源管理的功能
	page_cgroup_init();
	//使能对于物理页分配器的 debug 功能,该功能可以配置
	enable_debug_pagealloc();
	//内核内存泄露检测工具的初始化
	kmemleak_init();
	//初始化动态对象调试工具所需的内存,通过 kmem_cache 分配器创建
	debug_objects_mem_init();
	//ID 管理功能的所需内存初始化,同样是通过 kmem_cache 分配器创建
	//ID 管理功能是建立 ID 和指针的关联,比如 I2C 的从地址和管理结构的管理
	idr_init_cache();
	//为每个 CPU 分配一定的物理页
	setup_per_cpu_pageset();
	//非一致性内存策略初始化,功能和物理内存的分布相关,嵌入式通常不需要该功能
	numa_policy_init();
	//与体系结构相关的时间系统后半部的初始化函数,通常 ARM 下为空
	if (late_time_init)
		late_time_init();
	//sched_clock 主要是启动后的时间,以纳秒为单位,如记录 timestamp 功能。这里使能该功能
```

```
    sched_clock_init();
    //计算处理器的性能,可以依据指令延时估计,这里会输出如下的内容
    // Calibrating delay loop... 599.65 BogoMIPS (lpj=2998272)
    calibrate_delay();
    //对 PID 管理所需的内存进行分配
    pidmap_init();
    //对于匿名映射的 vma 管理实体分配内存,并初始化
    anon_vma_init();
#ifdef CONFIG_X86
    if (efi_enabled)
        efi_enter_virtual_mode();
#endif
    //thread_info 通常是在进程的内核栈低地址。通常为空
    thread_info_cache_init();
    //初始化 Credentials,这是在 Linux 内核中,用来管控权限的机制。通常用于检查用户的访问权限等
    cred_init();
    //初始化 fork 机制,参考 totalram_pages 决定可以创建进程数量 max_threads
    fork_init(totalram_pages);
    //初始化进程中各种资源(信号相关、文件系统相关、虚拟地址管理相关等)需要的内存
    proc_caches_init();
    //初始化文件系统 buffer 的管理空间
    buffer_init();
    //内核密匙管理系统初始化
    key_init();
    //内核安全框架初始化
    security_init();
    //内核调试系统后期初始化
    dbg_late_init();
    //初始化虚拟文件系统 VFS 需要的 dcache 和 inode cache 空间
    vfs_caches_init(totalram_pages);
    //信号管理的初始化,主要是初始化信号队列的空间
    signals_init();
    /* rootfs populating might need page-writeback */
    //初始化文件系统 writeback 回写机制,会产生 pdflush 内核线程来执行被修改的
    //页 Cache 的回写操作
    page_writeback_init();
#ifdef CONFIG_PROC_FS
    //初始化 proc 文件系统,主要用于查看和设置内核的信息
    proc_root_init();
#endif
    //内核资源管理功能的初始化
    cgroup_init();
```

```
    //对 CPU 资源管理功能的初始化
    cpuset_init();
    //任务统计网络发送功能的初始化
    taskstats_init_early();
    //用于统计 task 等待资源消耗时间的功能初始化
    delayacct_init();

    //检查处理器是否有特殊的 bug,bug 主要影响的是处理器 cache,tlb 和分支预测造成的 write
    //buffer 一致性的问题(原理是对通过不同虚拟地址映射的相同物理地址写入数值后,检查
    //最后的结果是否正确)
    check_bugs();

    //对高级控制和电源管理接口进行初始化,ARM 体系结构通常为空
    acpi_early_init(); /* before LAPIC and SMP init */
    sfi_init_late();

    //ftrace 提供对于内核的各种追踪功能
    ftrace_init();

    /* Do the rest non-__init'ed, we're now alive */
    //后续的内核初始化
    rest_init();
}
```

通过 start_kernel，可以对 Linux 内核的各个模块的功能有一个基本的了解。可以看到 Linux 内核还是提供了丰富的工具来帮助开发者调试和管理整个系统。

6. rest_init 和 kernel_init

基本的系统功能都已经初始化好，而且中断已经使能，下面就可以准备接收外部事件了。但是不要忘了，内核抢占还是 disable 的，并没有使能。此时系统中只有一个进程，其他进程是在 rest_init 中创建的，图 4-18 是 rest_init 的主要框架。

从图 4-18 中可见，Linux 内核中有三个最基本的任务进程，分别为 PID = 0（只是相当于并没有该 PID）的 idle 任务（就是之前提到的需要处理器进行特殊标记的任务）；PID = 1 的任务，负责初始化所有用户系统相关进程的 init 任务；以及 PID = 2 任务，负责产生内核线程的 kthreadd 任务。

当使用 ps 查看 Linux 所有进程时，会有如下信息。从中可见 PID 是从 1 开始的，而头两个任务就是在 rest_init 中创建的。

```
USER  PID %CPU %MEM   VSZ  RSS TTY  STAT START  TIME COMMAND
root    1  0.0  0.0 24732 2612 ?    Ss   09:27  0:00 /sbin/init
root    2  0.0  0.0     0    0 ?    S    09:27  0:00 [kthreadd]
```

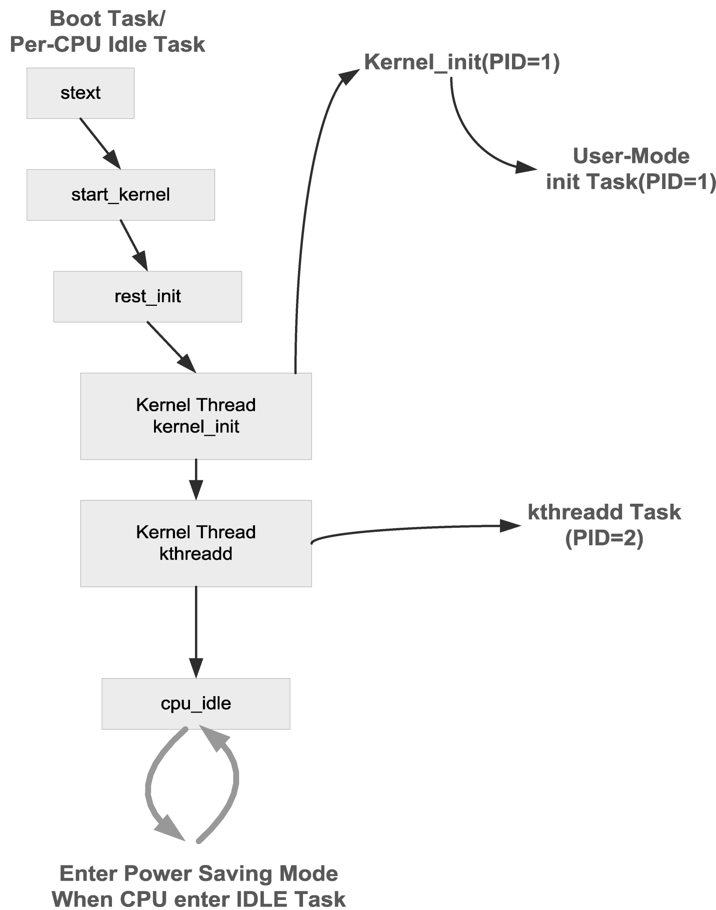


图 4-18　rest_init 框架图

接下来看看 rest_init 的代码：

```
static noinline void __init_refok rest_init(void)
{
    int pid;

    //启动 RCU 机制，之后就可以使用了
    rcu_scheduler_starting();
    /*
     * We need to spawn init first so that it obtains pid 1, however
     * the init task will end up wanting to create kthreads, which, if
     * we schedule it before we create kthreadd, will OOPS.
     */
    //创建 init 任务的内核线程(内核中资源都是可见，所以不需要进程的概念)，相应的 PID 是
    //1。内核线程会执行 kernel_init 来进行后续的操作
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    //设定 NUMA 的策略
```

```
    numa_default_policy( ) ;
    //创建 PID 为 2 的任务,kthreadd 主要用于创建内核线程
    pid = kernel_thread( kthreadd, NULL, CLONE_FS | CLONE_FILES) ;
    //下面是 RCU 保护区域
    rcu_read_lock( ) ;
    //获得 kthreadd 的 task_struct 结构,并保存在全局变量 kthread_task 中。这个全局变量作为
    //Linux 系统中守护进程的父进程。这是由于 init 进程拥有控制台,是普通进程的父进程,
    //而守护进程也需要父进程在其退出时为其释放资源
    kthreadd_task = find_task_by_pid_ns( pid, &init_pid_ns) ;
    rcu_read_unlock( ) ;
    //注意内核中要创建新的任务,通常需要 kthread_create 接口,保证干净的上下文。而 kthread_
    //create 则需要内核线程 kthreadd 来进行后续操作,所以这里由 complete 来完成同步,允许
    //init 进行后续操作
    complete( &kthreadd_done) ;

    / *
     * The boot idle thread must execute schedule( )
     * at least once to get things moving:
     * /
    //进程有不同的调度类型,而对于当前的进程,之前已经说明是作为 idle 进程,其调度应该
    //在所有进程都不在运行状态时才执行。这样就要将其归入特别的 idle 类型,该函数就是
    //执行这个操作
    init_idle_bootup_task( current) ;
    //现在已经有多个进程了,应该允许调度和抢占了,首先允许内核抢占
    preempt_enable_no_resched( ) ;
    //这里进行调度,由调度器决定是否切换到其他进程执行
    schedule( ) ;
    //到这里说明没有其他任务在运行状态,那么当前 CPU 要进行 idle 的操作,在进行 idle 操作
    //期间不能切换到其他进程执行,所以要禁止内核抢占
    preempt_disable( ) ;

    / * Call into cpu_idle with preempt disabled * /
    //执行 cpu_idle 操作,让处理器休息一下,函数内最终是死循环,切换到 idle 进程就休息一下
    //以达到节能的效果
    cpu_idle( ) ;
}
```

对 Linux 内核初始化讲到现在，仍然没有看到设备相关的初始化，各种设备的初始化是在哪里执行的呢？接下来看看 kernel_init 会有些什么发现。

```
static int __init kernel_init( void * unused)
{
    / *
     * Wait until kthreadd is all set - up.
```

```
  */
//由于后续操作需要通过 PID 为 2 的 kthreadd 创建内核线程,所以这里等待内核线程
//kthreadd 创建完成
wait_for_completion(&kthreadd_done);
/*
  * init can allocate pages on any node
  */
//init 进程的权限大,可以在所有的内存区域获得内存页,这里进行相应的设置
set_mems_allowed(node_states[N_HIGH_MEMORY]);
/*
  * init can run on any cpu.
  */
//同样 init 进程应该可以在所有的 CPU 上执行
set_cpus_allowed_ptr(current, cpu_all_mask);
/*
  * Tell the world that we' re going to be the grim
  * reaper of innocent orphaned children.
  *
  * We don' t want people to have to make incorrect
  * assumptions about where in the task array this
  * can be found.
 */
//Linux 系统中,对于孤儿进程的托管,是由 init 进程完成的,相应的设置就在这里
init_pid_ns.child_reaper = current;
//init 进程还是处理 ctrl-alt-delete 操作的进程,进行相应的设置
cad_pid = task_pid(current);
//和体系结构相关,为不同的 CPU 运行做准备
smp_prepare_cpus(setup_max_cpus);
//执行早期的 initcall,主要是段 .initcallearly.init 的函数,包括 workqueue 的初始化等需要早
//期的初始化操作
do_pre_smp_initcalls();
//这里激活 SMP 的其他处理器
smp_init();
//运行所有的 CPU 执行
sched_init_smp();
//接下来就是和设备及驱动相关的初始化执行了。只是对于 Linux 内核,将各个驱动的初始
//化放在了不同级别的 initcall 中进行顺序调用,相应的接口函数是 do_initcalls,当然在这之
//前还有必要的初始化(如设备模型)执行。后续会有详细介绍
do_basic_setup();

/* Open the /dev/console on the rootfs, this should never fail */
//后续通用进程都要打开的文件分别是标准输入输出和错误,均指向控制台
```

```
	if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) <0)
	printk(KERN_WARNING "Warning: unable to open an initial console. \n");

	(void) sys_dup(0);
	(void) sys_dup(0);
	/*
	 * check if there is an early userspace init. If yes, let it do all
	 * the work
	 */
	//如果需要进行早期的应用 init 操作
	if (! ramdisk_execute_command)
		ramdisk_execute_command = "/init";
	if (sys_access((const char __user *) ramdisk_execute_command, 0) ! =0) {
		ramdisk_execute_command = NULL;
		prepare_namespace();
	}

	/*
	 * Ok, we have completed the initial bootup, and
	 * we're essentially up and running. Get rid of the
	 * initmem segments and start the user-mode stuff..
	 */
	//进行后续的初始化,主要是在文件系统中查找最终的 init 程序,并执行。整个进程也会完
	//全被用户的程序替代。之前提到的对于只在初始化中需要的空间,如初始化函数的空间
	//等,都会在这里进行释放。这样系统才进入真正的执行状态
	init_post();
	return 0;
}
```

初始化时的 initcall 是比较繁杂的，涉及操作系统中的各个层面和功能，还和体系结构相关。为了方便查找和理解，笔者以 DM 3730 系列芯片内核为基础，列出了相应 initcall 的宏定义以及函数位置供读者参考。

```
#define early_initcall(fn)__define_initcall("early",fn,early)

	arch/arm/mach-omap2/omap4-common.c:early_initcall(omap_l2_cache_init);
	kernel/relay.c:early_initcall(relay_init);
	kernel/sched.c:early_initcall(migration_init);
	kernel/smp.c:early_initcall(init_call_single_data);
	kernel/softirq.c:early_initcall(spawn_ksoftirqd);
	kernel/stop_machine.c:early_initcall(cpu_stop_init);
```

```
    kernel/trace/trace.c:early_initcall(tracer_alloc_buffers);
    kernel/trace/trace_printk.c:early_initcall(init_trace_printk);
    kernel/trace/trace_workqueue.c:early_initcall(trace_workqueue_early_init);
    kernel/watchdog.c:early_initcall(spawn_watchdog_task);
    kernel/workqueue.c:early_initcall(init_workqueues);

#define pure_initcall(fn)__define_initcall("0",fn,0)
    drivers/cpufreq/cpufreq.c:pure_initcall(init_cpufreq_transition_notifier_list);
    net/core/net_namespace.c:pure_initcall(net_ns_init);

#define core_initcall(fn)__define_initcall("1",fn,1)
    arch/arm/mach-omap2/pm_bus.c:core_initcall(omap_pm_runtime_init);
    arch/arm/plat-omap/omap_device.c:core_initcall(omap_device_init);
    drivers/base/power/trace.c:core_initcall(early_resume_init);
    drivers/cpufreq/cpufreq.c:core_initcall(cpufreq_core_init);
    drivers/cpuidle/cpuidle.c:core_initcall(cpuidle_init);
    drivers/regulator/core.c:core_initcall(regulator_init);
    drivers/video/omap2/dss/core.c:core_initcall(omap_dss_init);
    fs/debugfs/inode.c:core_initcall(debugfs_init);
    kernel/cpu.c:core_initcall(alloc_frozen_cpus);
    kernel/power/hibernate.c:core_initcall(pm_disk_init);
    kernel/power/main.c:core_initcall(pm_init);
    kernel/power/swap.c:core_initcall(swsusp_header_init);
    kernel/sysctl.c:core_initcall(sysctl_init);
    kernel/time/jiffies.c:core_initcall(init_jiffies_clocksource);
    kernel/trace/trace_syscalls.c:core_initcall(init_ftrace_syscalls);
    mm/backing-dev.c:postcore_initcall(bdi_class_init);
    mm/memory.c:core_initcall(init_zero_pfn);
    net/core/netpoll.c:core_initcall(netpoll_init);
    net/core/sock.c:core_initcall(net_inuse_init);
    net/netlink/af_netlink.c:core_initcall(netlink_proto_init);
    net/socket.c:core_initcall(sock_init);  /* early initcall */

#define postcore_initcall(fn)__define_initcall("2",fn,2)
    drivers/base/node.c:postcore_initcall(register_node_type);
    drivers/char/tty_io.c:postcore_initcall(tty_class_init);
    drivers/char/vt.c:postcore_initcall(vtconsole_class_init);
    drivers/gpio/gpiolib.c:postcore_initcall(gpiolib_sysfs_init);
    drivers/i2c/i2c-core.c:postcore_initcall(i2c_init);
    drivers/spi/spi.c:postcore_initcall(spi_init);
    drivers/video/backlight/backlight.c:postcore_initcall(backlight_class_init);
    drivers/video/backlight/lcd.c:postcore_initcall(lcd_class_init);
```

```
    drivers/video/output. c:postcore_initcall( video_output_class_init) ;
    mm/backing - dev. c:postcore_initcall( bdi_class_init) ;

#define arch_initcall( fn) __define_initcall( "3" ,fn,3)
    arch/arm/kernel/atags. c:arch_initcall( init_atags_procfs) ;
    arch/arm/kernel/perf_event. c:arch_initcall( init_hw_perf_events) ;
    arch/arm/kernel/setup. c:arch_initcall( customize_machine) ;
    arch/arm/mach - omap2/clock2xxx. c:arch_initcall( omap2xxx_clk_arch_init) ;
    arch/arm/mach - omap2/clock3xxx. c:arch_initcall( omap3xxx_clk_arch_init) ;
    arch/arm/mach - omap2/devices. c:arch_initcall( omap2_init_devices) ;
    arch/arm/mach - omap2/mcbsp. c:arch_initcall( omap2_mcbsp_init) ;
    arch/arm/mach - omap2/pm - debug. c:arch_initcall( pm_dbg_init) ;
    arch/arm/plat - omap/32ksynctimer. c:arch_initcall( omap_init_clocksource_32k) ;
    arch/arm/plat - omap/cpu - omap. c:arch_initcall( omap_cpufreq_init) ;
    arch/arm/plat - omap/devices. c:arch_initcall( omap_init_devices) ;
    arch/arm/plat - omap/dma. c:arch_initcall( omap_init_dma) ;
    arch/arm/plat - omap/fb. c:arch_initcall( omap_init_fb) ;
    arch/arm/plat - omap/fb. c:arch_initcall( omap_init_fb) ;
    arch/arm/plat - omap/gpio. c:arch_initcall( omap_gpio_sysinit) ;
    drivers/dma/dmaengine. c:arch_initcall( dma_channel_table_init) ;
    drivers/dma/dmaengine. c:arch_initcall( dma_bus_init) ;
    drivers/video/omap2/vram. c:arch_initcall( omap_vram_init) ;
    sound/soc/omap/mcpdm. c:arch_initcall( omap_mcpdm_init) ;

#define subsys_initcall( fn) __define_initcall( "4" ,fn,4)
    arch/arm/mach - omap2/devices. c:subsys_initcall( omap_init_wdt) ;
    arch/arm/mach - omap2/emu. c:subsys_initcall( emu_init) ;
    drivers/char/misc. c:subsys_initcall( misc_init) ;
    drivers/connector/connector. c:subsys_initcall( cn_init) ;
    drivers/dma/ipu/ipu_idmac. c:subsys_initcall( ipu_init) ;
    drivers/gpio/gpiolib. c:subsys_initcall( gpiolib_debugfs_init) ;
    drivers/i2c/busses/i2c - omap. c:subsys_initcall( omap_i2c_init_driver) ;
    drivers/input/input. c:subsys_initcall( input_init) ;
    drivers/leds/led - class. c:subsys_initcall( leds_init) ;
    drivers/mfd/twl - core. c:subsys_initcall( twl_init) ;
    drivers/mmc/core/core. c:subsys_initcall( mmc_init) ;
    drivers/net/phy/phy_device. c:subsys_initcall( phy_init) ;
    drivers/power/power_supply_core. c:subsys_initcall( power_supply_class_init) ;
    drivers/regulator/twl - regulator. c:subsys_initcall( twlreg_init) ;
    drivers/rtc/class. c:subsys_initcall( rtc_init) ;
    drivers/spi/omap2_mcspi. c:subsys_initcall( omap2_mcspi_init) ;
    kernel/params. c:subsys_initcall( param_sysfs_init) ;
```

```
        kernel/power/poweroff.c:subsys_initcall(pm_sysrq_init);
        lib/raid6/algos.c:subsys_initcall(raid6_select_algo);
        net/core/dev.c:subsys_initcall(net_dev_init);
        net/core/sock.c:subsys_initcall(proto_init);
        sound/core/sound.c:subsys_initcall(alsa_sound_init);
        sound/sound_core.c:subsys_initcall(init_soundcore);

#define fs_initcall(fn) __define_initcall("5",fn,5)
        arch/arm/plat-omap/debug-leds.c:fs_initcall(fpga_init);
        drivers/base/firmware_class.c:fs_initcall(firmware_class_init);
        drivers/char/mem.c:fs_initcall(chr_dev_init);
        drivers/cpufreq/cpufreq_conservative.c:fs_initcall(cpufreq_gov_dbs_init);
        drivers/cpufreq/cpufreq_ondemand.c:fs_initcall(cpufreq_gov_dbs_init);
        drivers/cpufreq/cpufreq_performance.c:fs_initcall(cpufreq_gov_performance_init);
        drivers/cpufreq/cpufreq_powersave.c:fs_initcall(cpufreq_gov_powersave_init);
        drivers/cpufreq/cpufreq_userspace.c:fs_initcall(cpufreq_gov_userspace_init);
        drivers/usb/musb/musb_core.c:fs_initcall(musb_init);
        fs/cachefiles/main.c:fs_initcall(cachefiles_init);
        init/initramfs.c:rootfs_initcall(populate_rootfs);
        init/noinitramfs.c:rootfs_initcall(default_rootfs);
        kernel/time/clocksource.c:fs_initcall(clocksource_done_booting);
        kernel/trace/ftrace.c:fs_initcall(ftrace_init_debugfs);
        net/core/sysctl_net_core.c:fs_initcall(sysctl_core_init);
        net/ipv4/af_inet.c:fs_initcall(inet_init);

#define rootfs_initcall(fn) __define_initcall("rootfs",fn,rootfs)
        init/initramfs.c:rootfs_initcall(populate_rootfs);
        init/noinitramfs.c:rootfs_initcall(default_rootfs);

#define device_initcall(fn) __define_initcall("6",fn,6)
        arch/arm/mach-omap2/pm.c:device_initcall(omap2_common_pm_init);
        drivers/base/topology.c:device_initcall(topology_sysfs_init);
        drivers/video/omap2/dss/core.c:device_initcall(omap_dss_init2);

#define late_initcall(fn) __define_initcall("7",fn,7)
        arch/arm/kernel/crunch.c:late_initcall(crunch_init);
        arch/arm/kernel/thumbee.c:late_initcall(thumbee_init);
        arch/arm/kernel/traps.c:late_initcall(arm_mrc_hook_init);
        arch/arm/mach-omap2/mux.c:late_initcall(omap_mux_late_init);
        arch/arm/mach-omap2/pm24xx.c:late_initcall(omap2_pm_init);
        arch/arm/mach-omap2/pm34xx.c:late_initcall(omap3_pm_init);
        arch/arm/mach-omap2/pm44xx.c:late_initcall(omap4_pm_init);
```

```
    arch/arm/plat-omap/clock.c:late_initcall(clk_disable_unused);
    arch/arm/plat-omap/clock.c:late_initcall(clk_debugfs_init);
    drivers/base/power/trace.c:late_initcall(late_resume_init);
    drivers/block/hd.c:late_initcall(hd_init);
    drivers/char/random.c:late_initcall(seqgen_init);
    drivers/dma/dmatest.c:late_initcall(dmatest_init);
    drivers/regulator/core.c:late_initcall(regulator_init_complete);
    drivers/video/omap2/omapfb/omapfb-main.c:late_initcall(omapfb_init);
    kernel/kprobes.c:late_initcall(debugfs_kprobe_init);
    kernel/panic.c:late_initcall(init_oops_id);
    kernel/pm_qos_params.c:late_initcall(pm_qos_power_init);
    kernel/power/hibernate.c:late_initcall(software_resume);
    kernel/power/suspend_test.c:late_initcall(test_suspend);
    kernel/printk.c:late_initcall(printk_late_init);
    kernel/sched.c:late_initcall(sched_init_debug);
    kernel/taskstats.c:late_initcall(taskstats_init);
    kernel/trace/trace.c:late_initcall(clear_boot_tracer);
    mm/kmemleak.c:late_initcall(kmemleak_late_init);
    mm/page_alloc.c:late_initcall(fail_page_alloc_debugfs);
    mm/swapfile.c:late_initcall(max_swapfiles_check);
    net/core/dev.c:late_initcall_sync(initialize_hashrnd);
    net/core/drop_monitor.c:late_initcall(init_net_drop_monitor);
    net/ipv4/ipconfig.c:late_initcall(ip_auto_config);
```

从这些 initcall 中可以看出，很多功能模块、设备驱动框架和设备驱动的初始化操作都集中在这里进行了。当然这些初始化并不都执行，这和内核的配置相关，但是对了解明确各个模块的初始化流程还是很有帮助的。后面讲到具体驱动的时候会详细介绍相应的初始化操作。

7. 体系结构和板级相关的初始化

对 Linux 系统初始化有了整体的认识之后，现在回过头来看看体系结构相关和板级相关的初始化操作 setup_arch（ARM 体系结构下在 arch/arm/kernel/setup.c 中）。这其中的细节对于做系统移植还是十分重要的。以下是 DM 3730 内核 2.6.37 版本的相关 setup_ arch 代码，虽然版本有些老，但相应的机制是相同的。

```
void __init setup_arch(char **cmdline_p)
{
    struct tag *tags = (struct tag *)&init_tags;
    struct machine_desc *mdesc;
    char *from = default_command_line;
    //体系结构相关的,栈上调用链展开功能的初始化。该功能主要用于调试时查看栈上的函数
    //调用关系
    unwind_init();
```

```
	//再次检测处理器类型,并初始化处理器相关的底层变量。内核启动时的处理器信息就是通
	//过这个函数打印的。会输出如下信息
	//CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr = 10c53c7f
	//CPU: VIPT nonaliasing data cache, VIPT aliasing instruction cache
	//加载重要的数据结构 proc_info_list,其中包括了处理器内部 tlb、cache 等操作的具体接口。
	//具体的实现是在 arch/arm/mm/proc - *.S 中(处理器相关)
	setup_processor();

	//加载板级相关的信息。主要是针对板级特殊的信息和初始化实体。这里会对 machine_
	//arch_type 进行检查,而包括 machine_arch_type 的生成需要板级一系列操作。machine_arch_
	//type 的生成需要 arch/arm/tools/mach - types 中加入相应的信息,在 Kconfig 中加入相应的配
	//置,这样才能在对内核进行正确的配置后,生成正确的 machine_arch_type。在板级文件中则
	//通过宏 MACHINE_START 中的正确属性名形成正确的 machine_desc 属性定义,最终才能匹
	//配成功
	mdesc = setup_machine(machine_arch_type);
	machine_name = mdesc -> name;

	//设置板级重启的方式,是软重启还是硬重启。体系结构中都有重启的接口,如 ARM 的
	//arm_machine_restart,而其中会调用特别的处理器的接口函数 arch_reset(DM 3730 是定义
	//在 plat/system.h 中),通过该函数完成真正系统重启操作。DM 3730 的实现就是调用
	//PRCM 接口 rest 处理器
	if (mdesc -> soft_reboot)
		reboot_setup("s");

	//下面开始就是对 boot loader 传入的参数进行处理了。注意对于新的 device tree 架构实现这
	//部分会不相同
	if (__atags_pointer)
		tags = phys_to_virt(__atags_pointer);
	else if (mdesc -> boot_params)
		tags = phys_to_virt(mdesc -> boot_params);

#if defined(CONFIG_DEPRECATED_PARAM_STRUCT)
	/*
	 * If we have the old style parameters, convert them to
	 * a tag list.
	 */
	if (tags -> hdr.tag != ATAG_CORE)
		convert_to_tag_list(tags);
#endif
	if (tags -> hdr.tag != ATAG_CORE)
		tags = (struct tag *)&init_tags;
```

```
//如果需要则根据参数所在内存区域对系统内存信息进行修正,通常不需要
if (mdesc -> fixup)
	mdesc -> fixup(mdesc, tags, &from, &meminfo);

if (tags -> hdr.tag == ATAG_CORE) {
	if (meminfo.nr_banks != 0)
		squash_mem_tags(tags);
	save_atags(tags);
	parse_tags(tags);
}

//对于内核的 memory 管理实体,当然要知道内核的各个重要段的位置,这里对这些信息进
//行初始化
init_mm.start_code = (unsigned long) _text;
init_mm.end_code   = (unsigned long) _etext;
init_mm.end_data   = (unsigned long) _edata;
init_mm.brk        = (unsigned long) _end;

/* parse_early_param needs a boot_command_line */
strlcpy(boot_command_line, from, COMMAND_LINE_SIZE);

/* populate cmd_line too for later use, preserving boot_command_line */
strlcpy(cmd_line, boot_command_line, COMMAND_LINE_SIZE);
*cmdline_p = cmd_line;
//对需要尽早解析的内核参数进行解析,是通过 early_param 定义的参数,如 mem 等。对参数
//解析时会根据 mem 进行 meminfo 的设置
parse_early_param();

//在此处按地址数据从小到大排序 meminfo 中的数据,并初始化全局的内存块(memblock)数
//据。需要注意的是,板级特性有可能会保留一部分空间进行特别的管理和操作。machine_
//desc 中有保留空间的接口 reserve,会在这里被调用
arm_memblock_init(&meminfo, mdesc);

//建立页表,页表实际就是最终的映射。实际的系统还需要 zero 页和 bad 页来对一些地址进
//行特别的映射,实现一定的功能。另外板级特别的映射和向量表也在该处进行处理,通过
//devicemaps_init 实现。最后,初始化阶段的 bootmem 分配器也在这里进行初始化。对于地
//址映射的细节后续会进行讲解
paging_init(mdesc);
//对于地址这一资源要先将已有的资源请求出来,其中这里主要针对的是 iomem 进行请求,
//会包括板级视频显卡,PCI 以及系统内存等
request_standard_resources(&meminfo, mdesc);
```

```
#ifdef CONFIG_SMP
    //针对 SMP 处理器,初始化可能存在的 CPU 映射
    if (is_smp())
        smp_init_cpus();
#endif
    //通过内核参数中的 crashkernel,保留下用于主内核崩溃时获取内核信息的内存
    reserve_crashkernel();

    //初始化 CPU 及其栈。
    cpu_init();
    //对片内的 TCM 类型内存进行初始化。
    tcm_init();

    /*
     * Set up various architecture - specific pointers
     */
    //初始化各种板级相关的指针,后续初始化(已经在之前的内容介绍了)会使用
    arch_nr_irqs = mdesc -> nr_irqs;
    init_arch_irq = mdesc -> init_irq;
    system_timer = mdesc -> timer;
    //注意这里的 init_machine 是在 arch_initcall(见之前 initcall 的内容)的 customize_machine
    //中调用的
    init_machine = mdesc -> init_machine;

    //根据配置对控制台进行特别处理
#ifdef CONFIG_VT
#if defined(CONFIG_VGA_CONSOLE)
    conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
    conswitchp = &dummy_con;
#endif
#endif
    //这里进行向量表的初始化,主要是将 entry - armv.S 中的代码复制到相应的地址空间,使得
    //系统可以正常执行。向量表的实际物理页是在之前的 paging_init 执行中,通过 vectors = early_
    //alloc(PAGE_SIZE);进行分配的
    early_trap_init();
}
```

最后，再来看看板级相关的初始化接口的定义。以 DM 3730 EVM 板的代码为例，在 mach - omap2 目录下的 board - omap3evm.c 中实现。详细内容如下：

```
MACHINE_START(OMAP3EVM, "OMAP3 EVM")
        .boot_params  = 0x80000100,
    .map_io           = omap3_map_io,
```

```
	.reserve		= omap_reserve,
	.init_irq		= omap3_evm_init_irq,
	.init_machine	= omap3_evm_init,
	.timer		= &omap_timer,
MACHINE_END
```

这里的接口都是板级相关的接口，对于其中的说明以及在初始化中的使用，之前的内容已经说明，具体的实现这里就暂时放一放，会在对应的功能中进行详细介绍。需要说明的是 init_machine 接口，这个接口对于开发板是十分重要的。因为板子上所有的设备信息初始化都是通过该接口来实现的，其会在 arch_initcall 中被调用，通常早于设备驱动初始化。这样驱动初始化后就能正常使用设备了。函数具体的信息如下：

```
static void __init omap3_evm_init(void)
{
	omap3_evm_get_revision();

	if (get_omap3_evm_rev() >= OMAP3EVM_BOARD_GEN_2)
		omap3evm_twldata.vaux2 = &omap3evm_vaux2;
	else
		omap3evm_twldata.vusb = &omap3_evm_vusb;

	if (cpu_is_omap3630())
		omap3_mux_init(omap36x_board_mux, OMAP_PACKAGE_CBB);
	else
		omap3_mux_init(omap35x_board_mux, OMAP_PACKAGE_CBB);

	omap3_evm_i2c_init();

	platform_add_devices(omap3_evm_devices, ARRAY_SIZE(omap3_evm_devices));

	spi_register_board_info(omap3evm_spi_board_info,
				ARRAY_SIZE(omap3evm_spi_board_info));

	omap_serial_init();

	/* OMAP3EVM uses ISP1504 phy and so register nop transceiver */
	usb_nop_xceiv_register(0);

#ifndef CONFIG_MACH_FLASHBOARD
	if (get_omap3_evm_rev() >= OMAP3EVM_BOARD_GEN_2) {
		/* enable EHCI VBUS using GPIO22 */
		omap_mux_init_gpio(22, OMAP_PIN_INPUT_PULLUP);
		gpio_request(OMAP3_EVM_EHCI_VBUS, "enable EHCI VBUS");
		gpio_direction_output(OMAP3_EVM_EHCI_VBUS, 0);
```

```
            gpio_set_value(OMAP3_EVM_EHCI_VBUS, 1);

            /* Select EHCI port on main board */
            omap_mux_init_gpio(61, OMAP_PIN_INPUT_PULLUP);
            gpio_request(OMAP3_EVM_EHCI_SELECT, "select EHCI port");
            gpio_direction_output(OMAP3_EVM_EHCI_SELECT, 0);
            gpio_set_value(OMAP3_EVM_EHCI_SELECT, 0);

            /* setup EHCI phy reset config */
            omap_mux_init_gpio(21, OMAP_PIN_INPUT_PULLUP);
            ehci_pdata.reset_gpio_port[1] = 21;

            /* EVM REV >= E can supply 500mA with EXTVBUS programming */
            musb_board_data.power = 500;
            musb_board_data.extvbus = 1;
        } else {
            /* setup EHCI phy reset on MDC */
            omap_mux_init_gpio(135, OMAP_PIN_OUTPUT);
            ehci_pdata.reset_gpio_port[1] = 135;
        }
#else
        if (get_omap3_evm_rev() >= OMAP3EVM_BOARD_GEN_2) {
            /* EVM REV >= E can supply 500mA with EXTVBUS programming */
            musb_board_data.power = 500;
            musb_board_data.extvbus = 1;
        }
#endif
        usb_musb_init(&musb_board_data);
#ifndef CONFIG_MACH_FLASHBOARD
        usb_ehci_init(&ehci_pdata);
#endif
        ads7846_dev_init();
        omap3evm_init_smsc911x();
        omap3_evm_display_init();

#ifdef CONFIG_USB_ANDROID
        omap3evm_android_gadget_init();
#endif

        omap3_evm_pm_init();

        /* NAND */
        board_nand_init(omap3_evm_nand_partitions,
                ARRAY_SIZE(omap3_evm_nand_partitions),
                0, NAND_BUSWIDTH_16);
#ifndef CONFIG_MACH_FLASHBOARD
```

```
        board_onenand_init(omap3_evm_onenand_partitions,
                ARRAY_SIZE(omap3_evm_onenand_partitions), 0);
#endif
#ifdef CONFIG_WL12XX_PLATFORM_DATA
        /* WL12xx WLAN Init */
        if (wl12xx_set_platform_data(&omap3evm_wlan_data))
            pr_err("error setting wl12xx data\n");
        platform_device_register(&omap3evm_wlan_regulator);
#endif
#ifdef CONFIG_TI_ST
        omap3evm_init_btwilink();
#endif
#ifdef CONFIG_SND_SOC_WL1271BT
        wl1271bt_clk_setup();
#endif
}
```

从中可见初始化涉及芯片的封装类型、引脚复用和各种设备（包括 I^2C、SPI、电源、USB 等）。每种初始化细节留到相应功能和设备驱动部分时详述。

至此，包括核心处理器和板级的设备初始化的框架都进行了说明，整个初始化部分就比较完整了。

4.2 地址映射

4.2.1 地址映射的基本需求

Linux 内核中处理器使用的是虚拟地址，而无论内存还是物理设备的地址都是物理地址，这就需要进行虚拟地址到物理地址的映射。地址映射在内核的初始化阶段会输出类似如下的信息（DM 3730 EVM 内核的输出），从这些信息中可以了解内核地址映射的基本情况。

```
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (4 kB)
    fixmap  : 0xfff00000 - 0xfffe0000   (896 kB)
    DMA     : 0xffc00000 - 0xffe00000   (2 MB)
    vmalloc : 0xe0800000 - 0xf8000000   (376 MB)
    lowmem  : 0xc0000000 - 0xe0000000   (512 MB)
    modules : 0xbf000000 - 0xc0000000   (16 MB)
      .init : 0xc0008000 - 0xc002f000   (156 kB)
      .text : 0xc002f000 - 0xc0515000   (5016 kB)
      .data : 0xc0516000 - 0xc05514e0   (238 kB)
```

地址映射首先要符合体系结构的需求。毕竟每种体系结构都有自己特点，而地址本身就属于体系结构的一部分。比如 32 位和 64 位的系统地址空间都有数量级的差别，自然地址映

射是不同的。

在一个体系结构之下，要考虑的因素主要就是内存的容量。地址映射要能够支持各种容量的内存（当然首先要处理器能够访问到）。即使地址空间不够，如果物理上能够访问，也需要想办法访问更多的内存空间。

容量的需求是一方面，空间使用的需求则是要尽量减少分配不到的情况。这就需要在映射设计时考虑不同的映射方式（这部分和内存管理有相关性）。

在映射这个大的功能下，之前讨论的主要是对内存的使用，而对于操作系统来说更大的一部分是设备的使用，设备 IO 空间同样要进行映射。只有符合体系结构的映射才能够从处理器访问到设备，访问设备就要考虑设备本身的限制了，所以在做映射的时候要考虑设备的特殊需求。

总之，对于整个内核空间的地址映射设计就是在操作系统层面，综合考虑体系结构、内存需求、处理器、设备和内核效率的结果。

4.2.2 地址映射框架介绍

首先要明确，这里讲述的地址映射框架主要是指内核空间虚拟地址的使用和映射。mmap 等用户空间的映射相关功能会在内存管理和实际的设备驱动部分再进行介绍。

1. ARM 体系结构空间分配

对于地址映射框架，首先要考虑的就是体系结构。嵌入式系统通常是采用 ARM 的处理器，相应的地址映射框架自然就从 Linux 内核的 ARM 体系结构入手。相应的地址映射，内核大牛们已经为开发人员设计好地址空间的分布并制定了规范，相应的信息如下（摘自 Documents/arm/memory.tx 文件）。注意 ARM 体系结构和相应的 Linux 内核支持也是不断演进的，相应的地址映射也会随之进行一定的变化，但是通常都是增加相应的细化空间，而不是进行大的调整。

```
StartEndUse
--------------------------------------------------------------------------
ffff8000   ffffffff   copy_user_page / clear_user_page use.
                      For SA11xx and Xscale, this is used to
                      setup a minicache mapping.

ffff4000   ffffffff   cache aliasing on ARMv6 and later CPUs.

ffff1000   ffff7fff   Reserved.
                      Platforms must not use this address range.

ffff0000   ffff0fff   CPU vector page.
                      The CPU vectors are mapped here if the
                      CPU supports vector relocation (control
                      register V bit.)

fffe0000   fffeffff   XScale cache flush area. This is used
                      in proc - xscale. S to flush the whole data
                      cache. (XScale does not have TCM.)
```

```
fffe8000    fffeffff   DTCM mapping area for platforms with
                       DTCM mounted inside the CPU.

fffe0000    fffe7fff   ITCM mapping area for platforms with
                       ITCM mounted inside the CPU.

fff00000    fffdffff   Fixmap mapping region. Addresses provided
                       by fix_to_virt( ) will be located here.

ffc00000    ffefffff   DMA memory mapping region. Memory returned
                       by the dma_alloc_xxx functions will be
                       dynamically mapped here.

ff000000    ffbfffff   Reserved for future expansion of DMA
                       mapping region.

fee00000    feffffff   Mapping of PCI I/O space.  This is a static
                       mapping within the vmalloc space.

VMALLOC_START   VMALLOC_END - 1   vmalloc( ) / ioremap( ) space.
                       Memory returned by vmalloc/ioremap will
                       be dynamically placed in this region.
                       Machine specific static mappings are also
                       located here through iotable_init( ).
                       VMALLOC_START is based upon the value
                       of the high_memory variable, and VMALLOC_END
                       is equal to 0xff000000.

PAGE_OFFSET    HIGH_MEMORY - 1Kernel direct - mapped RAM region.
                       This maps the platforms RAM, and typically
                       maps all platform RAM in a 1:1 relationship.

PKMAP_BASE     PAGE_OFFSET - 1Permanent kernel mappings
                       One way of mapping HIGHMEM pages into kernel
                       space.

MODULES_VADDR   MODULES_END - 1   Kernel module space
                       Kernel modules inserted via insmod are
                       placed here using dynamic mappings.

00001000         TASK_SIZE - 1   User space mappings
                       Per - thread mappings are placed here via
                       the mmap( ) system call.

00000000    00000fff   CPU vector page / null pointer trap
                       CPUs which do not support vector remapping
                       place their vector page here.   NULL pointer
                       dereferences by both the kernel and user
                       space are also caught via this mapping.
```

32 位的 ARM 系统，整个的地址空间只有 4 GB，其中还要为用户应用保留大部分的地址空间，所以通常来说 32 位体系结构会保留 1 GB 左右的地址空间给内核使用。对于这1 GB 的空间，向量表的位置是不允许移动的。ARM 体系结构中向量表可以是低地址或者高地址两种模式，通常处理器都是使用高地址。其他的空间则根据需要进行设计。

从映射说明文档中可见，在 ARM 体系结构中，地址空间分配满足各种需求。其中包括加速内核地址换算（PAGE_OFFSET 到 high_memory－1）的内存区域 lowmem，该区域是线性映射（linear mapping）区域；VMALLOC_START 至 VMALLOC_END－1 的区域用于离散页的分配，可减少碎片，增加内存的使用率，另外这部分地址空间在新的内核中还涵盖了设备 IO 地址空间（ARM 是统一地址的体系结构），设备 IO 地址需要在启动时在该空间中保留一部分静态空间，作为后续的 IO 地址映射使用；超过地址空间的内存可以通过定义 HIGH-MEM 功能使用，PKMAP_BASE 至 PAGE_OFFSET－1 的空间进行临时的映射和使用；其他还有为设备特别保留的空间等。可见在整个地址空间分配上已经考虑得十分周详。当然上述的空间不都是必须使用的，还是要按照实际的需要进行配置，通常情况下 ARM 体系结构下的地址映射关系如图 4-19 所示。

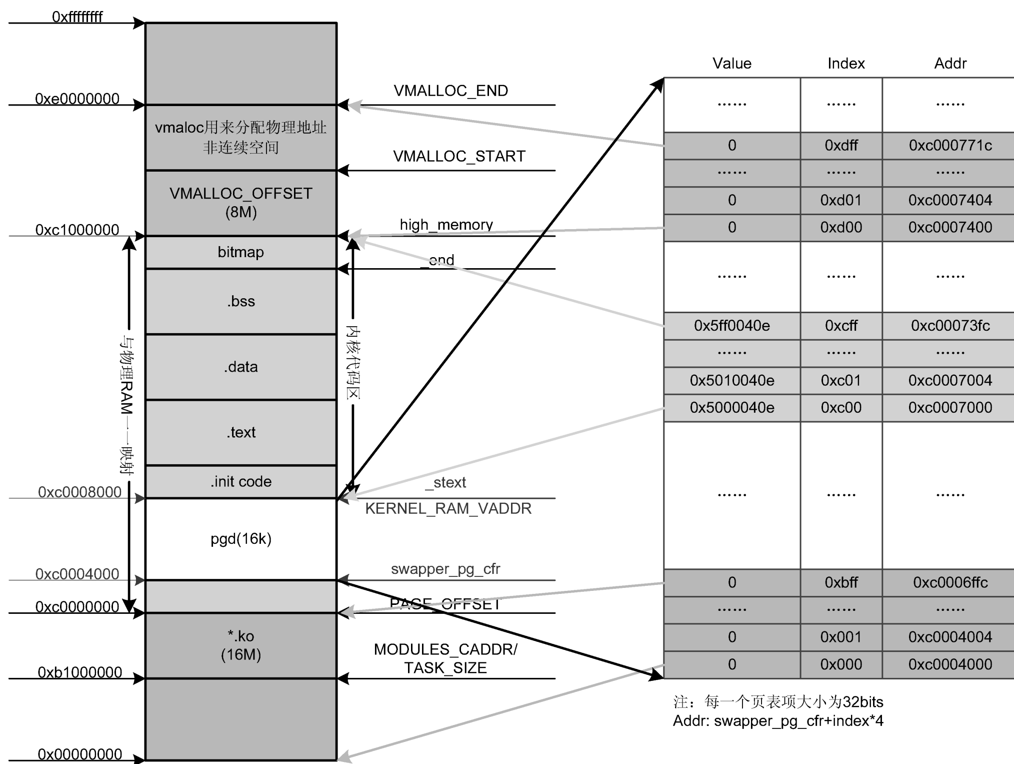


图 4-19　ARM 内核地址空间映射关系

2. 内核中的映射实现和相关的权限管理

地址映射的代码框架如图 4-20 所示。地址映射实际操作的接口是创建页表的 create_mapping 函数。ARM（统一地址体系结构的处理器）中无论是内存还是 IO 的映射最终都是通过 create_mapping 来实现的。而 iotable_init 则是提供给处理器的调用接口，用来进行 IO 空间的映射。

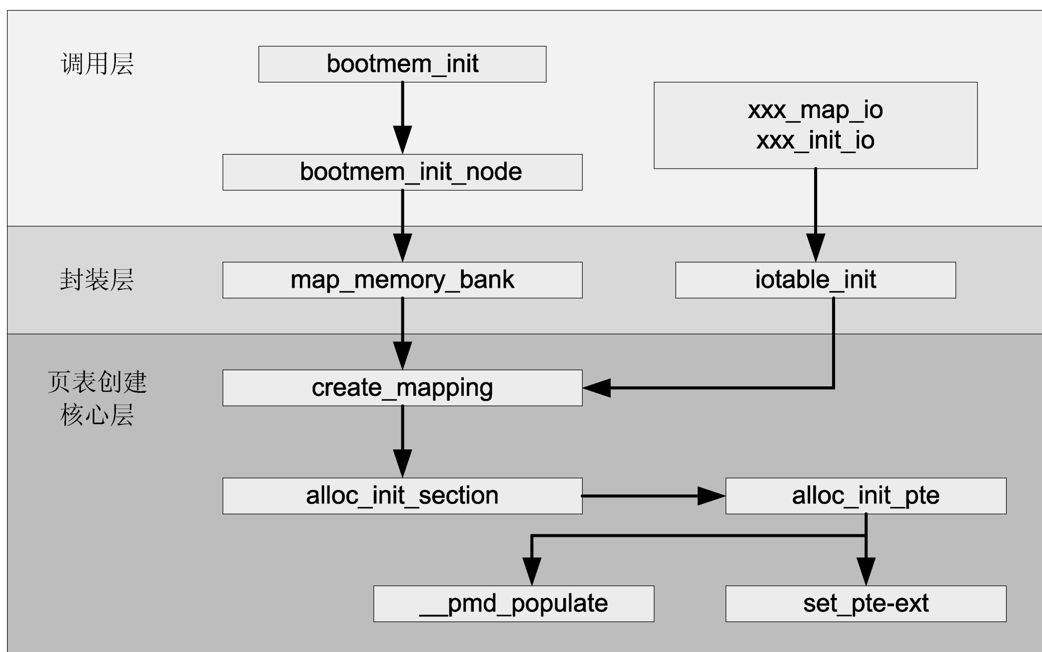


图 4-20　ARM 地址映射代码框架

谈到 create_mapping 的映射，就会涉及体系结构中不同的访问方式和权限。了解这些访问控制，对于理解内核的工作十分有帮助。

首先来看看 create_mapping 的传入参数的数据结构：

```
struct map_desc {
      unsigned long virtual;
      unsigned long pfn;
      unsigned long length;
      unsigned int type;
};
```

该结构的信息包括虚拟地址、物理地址、长度和类型。类型表示映射的种类和属性，如映射的是物理内存还是设备空间，相应的空间是否可以共享或者是否允许 Cache 等。具体的定义在两个文件中，内容如下：

```
arch/arm/include/asm/io.h
/*
 * Architecture ioremap implementation.
 */
#define MT_DEVICE                    0
#define MT_DEVICE_NONSHARED          1
#define MT_DEVICE_CACHED             2
#define MT_DEVICE_WC                 3
```

```
arch/arm/include/asm/mach/map.h
/* types 0 - 3 are defined in asm/io.h */
#define MT_UNCACHED             4
#define MT_CACHECLEAN           5
#define MT_MINICLEAN            6
#define MT_LOW_VECTORS          7
#define MT_HIGH_VECTORS         8
#define MT_MEMORY               9
#define MT_ROM                  10
```

系统中定义了这么多映射类型，最常用的是：MT_MEMORY，对应内存映射；MT_DEVICE，对应于通过 ioremap 的 IO 设备映射；MT_ROM，对应于 ROM；MT_LOW_VECTORS，对应 0 地址开始的向量；MT_HIGH_VECTORS，对应高地址开始的向量，通常设备都是采用高地址映射。对于向量的映射是由配置 CONFIG_VECTORS_BASE 的值来决定的。

create_mapping 的调用必然通过这些类型形成真正的页表属性才是有意义的。在相应的内核代码中 type = &mem_types[md -> type]；来将逻辑的类型转变为实际的映射页表的值。相应的 type 类型是 mem_type，具体内容如下：

```
struct mem_type {
        unsigned int prot_pte;
        unsigned int prot_l1;
        unsigned int prot_sect;
        unsigned int domain;
};
```

这里的值都是和访问权限、属性相关的，要了解详细的内容就需要从 ARM 相关的设计开始入手。首先，看看两级页表项的内容，分别如图 4-21 和图 4-22 所示。

	Bits 31 → 0 (fields from high to low)
Fault	Ignored | 0 | 0
Page table	Level 2 Descriptor Base Address | P | Domain | SBZ | 0 | 1
Section	Section Base Address | SBZ | 0 | nG | S | APX | TEX | AP | P | Domain | XN | C | B | 1 | 0
Supersection	Supersection Base Address | SBZ | | nG | S | APX | TEX | AP | P | Domain | XN | C | B | 1 | 0
Reserved	| 1 | 1

图 4-21　ARM 一级页表

一级页表和二级页表都会根据后两位有不同的映射形式。Linux 内核会针对之前提到的不同类型采用合适的形式进行映射，一级页表存放在 swapper_pg_dir 开始的 16 KB 区域内，在映射空间不满 1 MB 的情况下才使用二级页表。对 1 MB 空间采用 section 方式的只需要一次转换即可。

通过页表建立的映射关系如图 4-23 所示。

Fault	Ignored [31:2]									0	0
Large page	Large Page Base Address [31:16]	xN	TEX	nG	S	APX	SBZ	AP	C	B	0	1
Small page	Small Page Base Address [31:12]	nG	S	APX	SBZ	AP	C	B	1	xN	

图 4-22　ARM 二级页表

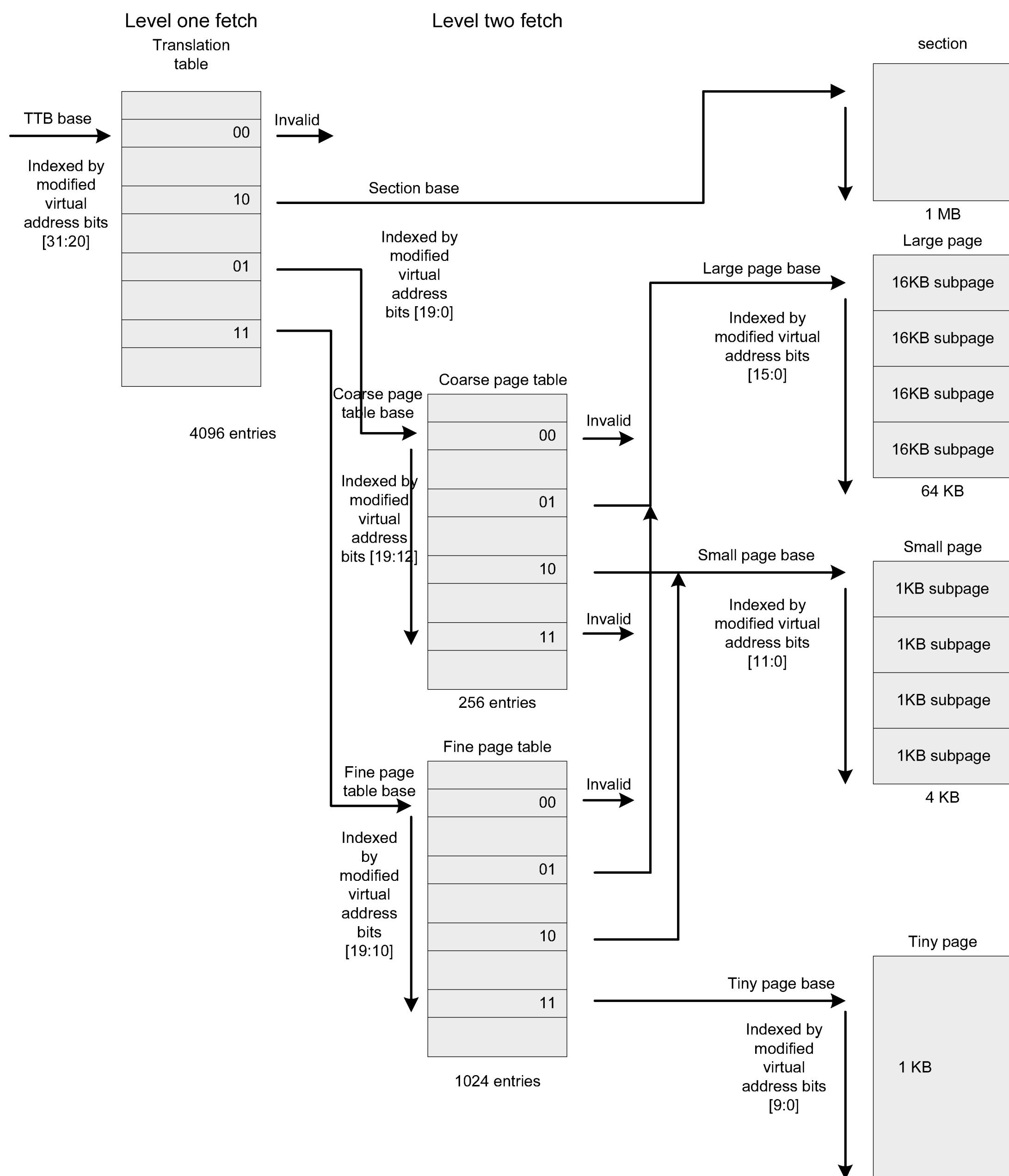


图 4-23　ARM 页表的映射关系

接下来对权限进行详细的说明，页表设置中访问权限相关的域分别为 Domain、AP 和 APX。这里的 Domain 是一个索引值，具体每个 Domain 的权限是由 CP15 的 C3 寄存器中的每两位来设定的。权限的具体内容见表 4-2。

表 4-2　Domain 权限具体内容

值	访问类型	含　义
0b00	无访问权限	此时访问该域将产生访问失效
0b01	用户（client）	根据 CP15 的 C1 控制寄存器中的 R 和 S 位以及页表中地址变换条目中的访问权限控制位 AP 来确定是否允许各种系统工作模式的存储访问
0b10	保留	使用该值会产生不可预知的结果
0b11	管理者（Manager）	不考虑 CP15 的 C1 控制寄存器中的 R 和 S 位以及页表中地址变换条目中的访问权限控制位 AP，在这种情况下，不管系统工作在特权模式，还是用户模式，都不会产生访问失效

Linux 内核只是用了几个 Domain，对应的权限值定义在 arch/arm/include/asm/domain. h 中，具体内容如下：

```
#define DOMAIN_KERNEL       0
#define DOMAIN_TABLE        0
#define DOMAIN_USER         1
#define DOMAIN_IO           2
…
#define DOMAIN_NOACCESS     0
#define DOMAIN_CLIENT       1
#define DOMAIN_MANAGER      3
```

AP 和 APX 的属性说明见表 4-3。通常使用 APX = 0 进行设置。

表 4-3　访问权限属性说明

APX	AP[1:0]	特权模式访问权限	用户模式访问权限
0	b00	禁止访问；S = 1，R = 0 或 S = 0，R = 1 时只读	禁止访问；S = 1，R = 0 时只读
0	b01	读写	禁止访问
0	b10	读写	只读
0	b11	读写	读写
1	b00	保留	保留
1	b01	只读	禁止访问
1	b10	只读	只读
1	b11	只读	只读

回头看看 mem_type 中的几个域现在就清楚了，prot_pte 是二级页表的访问控制属性，prot_l1 是有二级页表情况下的一级页表访问控制属性，prot_sect 是一级页表 section 模式访问控制属性，domain 代表一级页表访问属性中的 Domain，会根据需要填入到 prot_l1 或者 prot_sect 中。

Linux 内核中页表属性的位域定义在 arch/arm/include/asm/pgtable-hwdef. h 中具体如下：

```
#define PMD_TYPE_TABLE          (1 << 0)
#define PMD_TYPE_SECT           (2 << 0)
#define PMD_BIT4                (1 << 4)
#define PMD_DOMAIN(x)           ((x) << 5)
……
#define PMD_SECT_AP_WRITE       (1 << 10)
#define PMD_SECT_AP_READ        (1 << 11)
```

由于 domain 在访问权限判断中优先级高，所以它的设置就显得比较重要，Linux 对不同 domain 设置的相关代码如下：

```
arch/arm/include/asm/domain.h
#define domain_val(dom,type)    ((type) << (2 * (dom)))
…
#define set_domain(x)                                   \
        do {                                            \
        __asm__ __volatile__(                           \
        "mcr    p15, 0, %0, c3, c0      @ set domain"   \
          : : "r"(x));                                  \
        isb();                                          \
        } while(0)
#define modify_domain(dom,type)                                 \
        do {                                                    \
        struct thread_info *thread = current_thread_info();     \
        unsigned int domain = thread->cpu_domain;               \
        domain &= ~domain_val(dom, DOMAIN_MANAGER);             \
        thread->cpu_domain = domain | domain_val(dom, type);    \
        set_domain(thread->cpu_domain);                         \
        } while(0)

arch/arm/kernel/head.S
    ……
    mov     r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | \
                  domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | \
                  domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | \
                  domain_val(DOMAIN_IO, DOMAIN_CLIENT))
    mcr     p15, 0, r5, c3, c0, 0       @ load domain access register
    mcr     p15, 0, r4, c2, c0, 0       @ load page table pointer
    b       __turn_mmu_on
ENDPROC(__enable_mmu)
```

可见，内核在启动的初期就对这三个域的访问控制进行了设置。这里的设置主要是因为系统在内核态拥有特权级别，这样的设置可以减少权限检测。但这个设置并不是一成不变

的，内核提供了 modify_domain 的宏，作为修改域访问控制位的接口。内核在 setup_arch 中调用 early_trap_init，利用该接口将 DOMAIN_USER 的权限位设置成 DOMAIN_CLIENT。另外对于 DOMAIN_KERNEL 同样需要在内核执行用户态任务之前将其设置为 DOMAIN_CLIENT，这样才能进行访问权限的检查。对于 DOMAIN_KERNEL 的修改会通过 set_fs 接口来执行，这主要是因为系统调用，特别是文件系统的操作，经常会调用内核空间执行相应系统调用的接口，这些接口是与体系结构无关的，所以要进行地址检查，确保操作的数据来自用户空间而不是内核空间，如果从内核直接调用就会报告异常。这时候的解决方案就是临时提高权限，等操作完成后再恢复原有的权限，这样就有 set_fs 接口。ARM 体系结构中，由于有 ldrt（无论处理器当前的权限级别都以用户级权限执行）指令，像 get_user 这种获得用户数据的操作，自然用 ldrt 更合理。这样对前述的内核态参数的访问，使用 ldrt 时会有同样问题，所以在 ARM 体系结构下，会将体系结构无关的权限升级与 DOMAIN_KERNEL 的权限升级结合到一起。当然平时都是在 DOMAIN_CLIENT 的状态下进行权限检查的。另外在任务切换时同样需要进行 domain 权限的转换，这就由 thread_info 中的 cpu_domain 域记录并完成。这样就对整个系统任何时候都进行了合适的 domain 值的设置。做这么多的工作都是因为映射本身涉及权限，权限是和安全相关的，所以要格外的小心，哪怕花费精力也是必要的。

内核中各种映射类型及访问权限相关的设置见表 4-4。

表 4-4　映射类型和访问权限设置

内存映射类型	域定义	段页表项权限定义	L1 页表项权限定义	PTE 项权限定义
MT_DEVICE	DOMAIN_IO	PROT_SECT_DEVICE PMD_SECT_S	PMD_TYPE_TABLE	PROT_PTE_DEVICE L_PTE_MT_DEV_SHARED L_PTE_SHARED
MT_DEVICE_NONSHARED	DOMAIN_IO	PROT_SECT_DEVICE	PMD_TYPE_TABLE	PROT_PTE_DEVICE L_PTE_MT_DEV_NONSHARED
MT_DEVICE_CACHED	DOMAIN_IO	PROT_SECT_DEVICE PMD_SECT_WB	PMD_TYPE_TABLE	PROT_PTE_DEVICE L_PTE_MT_DEV_CACHED
MT_DEVICE_WC	DOMAIN_IO	PROT_SECT_DEVICE	PMD_TYPE_TABLE	ROT_PTE_DEVICE L_PTE_MT_DEV_WC
MT_UNCACHED	DOMAIN_IO	PMD_TYPE_SECT PMD_SECT_XN	PMD_TYPE_TABLE	PROT_PTE_DEVICE
MT_CACHECLEAN	DOMAIN_KERNEL	PMD_TYPE_SECT PMD_SECT_XN		
MT_MINICLEAN	DOMAIN_KERNEL	PMD_TYPE_SECT PMD_SECT_XN PMD_SECT_MINICACHE,		
MT_LOW_VECTORS	DOMAIN_USER		PMD_TYPE_TABLE	L_PTE_PRESENT L_PTE_YOUNG L_PTE_DIRTY L_PTE_EXEC
MT_HIGH_VECTORS	DOMAIN_USER		PMD_TYPE_TABLE	L_PTE_PRESENT L_PTE_YOUNG L_PTE_DIRTY L_PTE_USER L_PTE_EXEC

（续）

内存映射类型	域定义	段页表项权限定义	L1 页表项权限定义	PTE 项权限定义
MT_MEMORY	DOMAIN_KERNEL	PMD_TYPE_SECT PMD_SECT_AP_WRITE		
MT_ROM	DOMAIN_KERNEL	PMD_TYPE_SECT		

这样整个 ARM 体系结构的映射框架就比较清晰了。

3. 总结

Linux 内核 ARM 体系结构下整个空间映射的数据和权限的特点为 Linux 在 ARM 体系结构下通常是使用 4KB 大小的页，但是对于线性映射的内核空间（low memory）以及 IO 空间使用 1MB 大小的页。内核针对不同的应用场景一共有六种不同类型页映射，分别如下：

- 对于存放应用或共享库代码的页面，映射时会标记为只读，并且允许映射到多个应用的地址空间。
- 对于保护可写数据的页面要标记 XN 位表示不可执行，用于捕获试图执行数据区的错误。
- 对于用户用来映射普通文件的页面、包含 stack 或者 heap 的页面、kernel modules 使用的页面、内核线性映射的页面以及 vmalloc 空间的页面都被标记为“normal，cacheable”。而包含 stack 或者 heap 的页面在初始分配时设计成会引发缺页异常的只读零页，只有在首次写数据引发的缺页异常中才进行真正的分配操作。注意所有的页表项都是通过内核的线性映射空间进行分配的。
- 包含设备文件映射的页面由设备驱动负责映射。
- 包含异常向量表的页面被标记为 normal cacheable 并且用户态只读。
- 内核中静态和动态映射的设备存储区域需要设置为用户态不能访问。

熟悉这些类型对于理解内存管理是很有帮助的，因为这些类型也是内存页面使用的不同方式。

4.2.3 TI 芯片地址映射相关实现详解

1. 芯片相关的内核地址空间分配和映射

现在看看具体芯片在地址映射这部分是如何实现的。内存部分的映射完全可以通过启动参数以及一些宏定义实现。通过 mem 和 vmalloc 这两个内核提供的启动参数以及 VMALLOC_START 和 VMALLOC_END 两个宏，可以将 lowmem 和 vmalloc 的空间定下来。相关的宏定义 DM 3730 和 DM 816X 的内核代码如下：

```
arch/arm/plat-omap/include/mach/vmalloc.h
#define VMALLOC_END    0xf8000000UL

arch/arm/include/asm/pgtable.h
#ifndef VMALLOC_START
#define VMALLOC_OFFSET         (8*1024*1024)
#define VMALLOC_START      (((unsigned long)high_memory + VMALLOC_OFFSET) & ~(VMAL-
LOC_OFFSET-1))
#endif
```

vmalloc 的空间是在 lowmem 之后的 8 MB 开始的，当然 vmalloc 的空间不是随意大小的，

会有最小空间的限制，默认限制如下：

```
static void * __initdata vmalloc_min = (void *)(VMALLOC_END - SZ_128M);
```

当然相应的大小可以由启动参数 vmalloc 进行设置，但是也不能小于 16 MB。有了总的内存设置（启动参数 mem），有了 vmalloc 空间的结束设置和最小空间限制，内存的映射基本就解决了。如果内存容量超出了这些空间，还有 pkmap 可以进行映射。当然之前说明新的内核中（DM 3730 和 DM 816X 没有相应版本的实现）VMALLOC_END 已经直接由 ARM 内核设定为 0xff000000 了，这样的调整对于实现没有太大的影响，因为这是地址空间的调整，具体映射的内容可以在映射模块内部解决兼容性的问题。

2. IO 空间在内核中的映射

接下来的重点是 IO 空间的映射。首先，来看看 DM 3730 的物理空间是如何设置的，如图 4-24所示。图 4-24 引自《DM 3730 芯片手册》中第 204 页的表，由于表比较大，这里只截取一部分。

Quarter	Device Name	Start Address (Hex)	End Address (Hex)	Size	Description
Q0 (1GB)	**Boot space[1] GPMC**			**1MB 1GB or 1GB-1MB**	
	GPMC	0x0000 0000	0x3FFF FFFF	1GB	8/16 Ex[2]/R/W
Q1 (1GB)	**On-chip memory**			**128MB**	**ROM/SRAM address space**
	Boot ROM internal[1]	0x4000 0000	0x4001 3FFF	80KB	Reserved for boot code Not accessible after boot
		0x4001 4000	0x4001 BFFF	32KB	32-bit Ex[2]/R
	Reserved	0x4001 C000	0x400F FFFF	912KB	Reserved
	Reserved	0x4010 0000	0x401F FFFF	1MB	Reserved
	SRAM internal	0x4020 0000	0x4020 FFFF	64KB	32-bit Ex[2]/RW
	Reserved	0x4021 0000	0x4024 FFFF	256KB	Reserved
	Reserved	0x4025 0000	0x47FF FFFF	128,704KB	Reserved
	L4 interconnects			**128MB**	**All system peripherals**
	L4-Core	0x4800 0000	0x48FF FFFF	16MB	See Table 2-3.
	(L4-Wakeup)[3]	(0x4830 0000)	(0x4833 FFFF)	(256KB)	See Table 2-4.
	L4-Per	0x4900 0000	0x490F FFFF	1MB	See Table 2-5.
	Reserved	0x4910 0000	0x4FFF FFFF	111MB	Reserved
	SGX			**64MB**	**Graphic accelerator slave port**
	SGX	0x5000 0000	0x5000 FFFF	64KB	Graphic accelerator slave port
	Reserved	0x5001 0000	0x53FF FFFF	65,472KB	Reserved
	L4 emulation			**64MB**	**Emulation**
	L4-Emu	0x5400 0000	0x547F FFFF	8MB	See Table 2-6.
	Reserved	0x5480 0000	0x57FF FFFF	56MB	Reserved
	Reserved			**64MB**	**Reserved**
	Reserved	0x5800 0000	0x5BFF 0FFF	64MB	Reserved
	IVA2.2 subsystem			**64MB**	**IVA2.2 subsystem**
	IVA2.2 subsystem	0x5C00 0000	0x5EFF FFFF	48MB	IVA2.2 subsystem. See Table 2-8.
	Reserved	0x5F00 0000	0x5FFF FFFF	16MB	Reserved
	Reserved			**128MB**	**Reserved**
	Reserved	0x6000 0000	0x67FF FFFF	128MB	Reserved
	L3 interconnect			**128MB**	**Control registers**
	L3 control registers	0x6800 0000	0x68FF FFFF	16MB	See Table 2-2.
	Reserved	0x6900 0000	0x6BFF FFFF	48MB	Reserved
	SMS registers	0x6C00 0000	0x6CFF FFFF	16MB	Configuration registers SMS address space 2
	SDRC registers	0x6D00 0000	0x6DFF FFFF	16MB	Configuration registers SMS address space 3
	GPMC registers	0x6E00 0000	0x6EFF FFFF	16MB	Configuration registers GPMC address space 1
	Reserved	0x6F00 0000	0x6FFF FFFF	16MB	Reserved
	SDRC/SMS			**256MB**	**SDRC/SMS**

图 4-24　DM 3730 物理空间设置

IO 空间映射的目的就是将这些物理地址映射到整个内核的地址空间中。下面来看看 DM 3730 内核中映射关系是怎样的。在 arch/arm/plat-omap/include/mach/io.h 中定义映射关系如下：

```
/*
 * ----------------------------------------------------------------------------
 * Omap3 specific IO mapping
 * ----------------------------------------------------------------------------
 */

/* We map both L3 and L4 on OMAP3 */
#define L3_34XX_PHYS      L3_34XX_BASE/* 0x68000000 -->0xf8000000 */
#define L3_34XX_VIRT      (L3_34XX_PHYS + OMAP2_L3_IO_OFFSET)
#define L3_34XX_SIZE      SZ_1M   /* 44kB of 128MB used, want 1MB sect */

#define L4_34XX_PHYS      L4_34XX_BASE/* 0x48000000 -->0xfa000000 */
#define L4_34XX_VIRT      (L4_34XX_PHYS + OMAP2_L4_IO_OFFSET)
#define L4_34XX_SIZE      SZ_4M   /* 1MB of 128MB used, want 1MB sect */

/*
 * Need to look at the Size 4M for L4.
 * VPOM3430 was not working for Int controller
 */

#define L4_PER_34XX_PHYS   L4_PER_34XX_BASE
                           /* 0x49000000 -->0xfb000000 */
#define L4_PER_34XX_VIRT      (L4_PER_34XX_PHYS + OMAP2_L4_IO_OFFSET)
#define L4_PER_34XX_SIZE      SZ_1M

#define L4_EMU_34XX_PHYS   L4_EMU_34XX_BASE
                           /* 0x54000000 -->0xfe800000 */
#define L4_EMU_34XX_VIRT      (L4_EMU_34XX_PHYS + OMAP2_EMU_IO_OFFSET)
#define L4_EMU_34XX_SIZE      SZ_8M

#define OMAP34XX_GPMC_PHYSOMAP34XX_GPMC_BASE
                           /* 0x6e000000 -->0xfe000000 */
#define OMAP34XX_GPMC_VIRT   (OMAP34XX_GPMC_PHYS + OMAP2_L3_IO_OFFSET)
#define OMAP34XX_GPMC_SIZE      SZ_1M

#define OMAP343X_SMS_PHYSOMAP343X_SMS_BASE
                           /* 0x6c000000 -->0xfc000000 */
#define OMAP343X_SMS_VIRT(OMAP343X_SMS_PHYS + OMAP2_L3_IO_OFFSET)
```

```
#define OMAP343X_SMS_SIZESZ_1M

#define OMAP343X_SDRC_PHYSOMAP343X_SDRC_BASE
                        /* 0x6D000000 -->0xfd000000 */
#define OMAP343X_SDRC_VIRT     (OMAP343X_SDRC_PHYS+OMAP2_L3_IO_OFFSET)
#define OMAP343X_SDRC_SIZE     SZ_1M

  /* 3430 IVA - currently unmapped */
```

可见映射是从 0xf8000000 开始的，正好是在 vmalloc 之后，这样就不会有空间重叠的问题。从物理地址的角度和图 4-24 中描述的地址也是吻合的，只是代码中每部分的空间只保留使用的部分，不像物理上预留了更多的空间。

有了这样的映射关系之后，就要解决两个问题：一个是 ioremap 如何返回正确的虚拟地址；另外一个就是实际的映射建立。

首先来看 ioremap 虚拟地址返回是如何实现的。在 ARM 的 io.h 中有如下定义：

```
#ifndef __arch_ioremap
#define __arch_ioremap          __arm_ioremap
#define __arch_iounmap          __iounmap
#endif

#define ioremap(cookie,size)     __arch_ioremap((cookie),(size), MT_DEVICE)
#define ioremap_nocache(cookie,size)   __arch_ioremap((cookie),(size), MT_DEVICE)
#define ioremap_cached(cookie,size)    __arch_ioremap((cookie),(size), MT_DEVICE_CACHED)
#define ioremap_wc(cookie,size)        __arch_ioremap((cookie),(size), MT_DEVICE_WC)
#define iounmap                        __arch_iounmap
```

可见，可以通过定义特别的__arch_ioremap 来实现 ioremap 的重定向。DM 3730 等 TI 芯片就是通过该方法来实现该功能的。下面来看看细节，在 arch/arm/plat-omap/include/mach/io.h 中有如下定义：

```
#define __arch_ioremapomap_ioremap
#define __arch_iounmapomap_iounmap
```

最终由 omap_ioremap 来实现 ioremap 的功能，看看 omap_ioremap 的代码就清楚了。代码如下：

```
void __iomem *omap_ioremap(unsigned long p, size_t size, unsigned int type)
{
...
#ifdef CONFIG_ARCH_OMAP3
    if(cpu_is_omap34xx()){
        if(BETWEEN(p, L3_34XX_PHYS, L3_34XX_SIZE))
            return XLATE(p, L3_34XX_PHYS, L3_34XX_VIRT);
```

```
            if(BETWEEN(p, L4_34XX_PHYS, L4_34XX_SIZE))
                return XLATE(p, L4_34XX_PHYS, L4_34XX_VIRT);
            if(BETWEEN(p, OMAP34XX_GPMC_PHYS, OMAP34XX_GPMC_SIZE))
                return XLATE(p, OMAP34XX_GPMC_PHYS, OMAP34XX_GPMC_VIRT);
            if(BETWEEN(p, OMAP343X_SMS_PHYS, OMAP343X_SMS_SIZE))
                return XLATE(p, OMAP343X_SMS_PHYS, OMAP343X_SMS_VIRT);
            if(BETWEEN(p, OMAP343X_SDRC_PHYS, OMAP343X_SDRC_SIZE))
                return XLATE(p, OMAP343X_SDRC_PHYS, OMAP343X_SDRC_VIRT);
            if(BETWEEN(p, L4_PER_34XX_PHYS, L4_PER_34XX_SIZE))
                return XLATE(p, L4_PER_34XX_PHYS, L4_PER_34XX_VIRT);
            if(BETWEEN(p, L4_EMU_34XX_PHYS, L4_EMU_34XX_SIZE))
                return XLATE(p, L4_EMU_34XX_PHYS, L4_EMU_34XX_VIRT);
        }
#endif
...
#ifdef CONFIG_ARCH_TI81XX
        if(cpu_is_ti81xx()){
            if(BETWEEN(p, L4_SLOW_TI81XX_PHYS, L4_SLOW_TI81XX_SIZE))
                return XLATE(p, L4_SLOW_TI81XX_PHYS, L4_SLOW_TI81XX_VIRT);
            if(BETWEEN(p, TI81XX_L2_MC_PHYS, TI81XX_L2_MC_SIZE))
                return XLATE(p, TI81XX_L2_MC_PHYS, TI81XX_L2_MC_VIRT);
        }
#endif
        return __arm_ioremap_caller(p, size, type, __builtin_return_address(0));
}
```

其中所做的工作就是，如果是映射关系中的物理地址就直接转换为相应的虚拟地址并返回，否则就使用 ARM 内核提供的__arm_ioremap_caller 进行映射。注意__arm_ioremap_caller 使用的是 vmalloc 的空间进行映射，在相应版本内核中 vmalloc 的空间和 TI 芯片的 IO 映射空间是完全分开的，所以这里要直接返回相应的虚拟地址。新内核的 iotable_init 是通过 vmalloc 空间的静态分配来将这部分地址在初始化的时候保留，这样就可以直接使用 ARM 内核的 ioremap 而不需要进行重定向了。新内核的该功能进一步减少了不同处理器之间的差别，减少了移植的工作量。

具体映射的建立则是从板级相关的初始化接口 map_io 开始。对于 DM 3730 相应的 map_io 为 omap3_map_io，其内容如下：

```
void __init omap3_map_io(void)
{
        omap2_set_globals_3xxx();
        omap34xx_map_common_io();
}
```

重点是在 omap34xx_map_common_io 中，该函数及相关函数详细说明。其代码如下：

```
void __ init omap34xx_map_common_io( void)
{
    //设置相应的 IO 寄存器所属的 memory 地址的页表
    //io_desc 是根据芯片手册的 memory map 表定义的,占用尽量少的空间
    iotable_init( omap34xx_io_desc, ARRAY_SIZE( omap34xx_io_desc) ) ;

    //该函数根据 chip type 设定相关的特性 flag,并初始化片内 sram
    _omap2_map_common_io( ) ;
}

//本函数检查 omap chip 的版本特性,并进行 internal sram 的初始化
//包括根据 omap chip 的 type 设置 sram 地址及 kernel 的内部映射
static void __ init _omap2_map_common_io( void)
{
    / *  Normally devicemaps_init( ) would flush caches and tlb after
      *  mdesc -> map_io( ) , but we must also do it here because of the CPU
      *  revision check below.
      * /
    //通常 devicemaps_init 在调用 machine_desc 的 map_io 之后都会 flush tlb 和 cache。此处在 map_io
    //内部由于需要访问寄存器,为保险 flush 一下
    local_flush_tlb_all( ) ;
    flush_cache_all( ) ;

    omap2_check_revision( ) ;


    //初始化 on - chip sram
    omap_sram_init( ) ;
}
```

IO 空间页表的映射是通过 ARM 内核提供的 iotable_init 实现的。映射之后就进行寄存器的访问以及片内 RAM 的初始化，片内 RAM 的初始化由于和内存管理相关，还是放在内存管理的实现中进行说明。而对于具体的映射属性是通过 omap34xx_io_desc 数组定义的，具体的数组内容可以通过实际代码来了解细节。

4.3 中断处理

中断是指硬件停止当前正在运行的程序，将系统转向处理其他工作的能力。中断作为一种技术在处理器实现后就完全改变了整个系统的编程方式，为系统带来了外部事件。不要小看这些外部事件，从系统论的角度，一个系统只有不断有新鲜事物的进入才能得到发展，所谓“流水不腐户枢不蠹”。虽然中断带来的外部事件和新鲜事物无法同日而语，但是中断可以说是计算机系统的一次革命，给计算机系统带来了无限可能。

4.3.1　中断的基本需求

对于中断系统的需求，首先要看看中断在系统中的硬件连接是怎么样的，如图 4-25 所示。

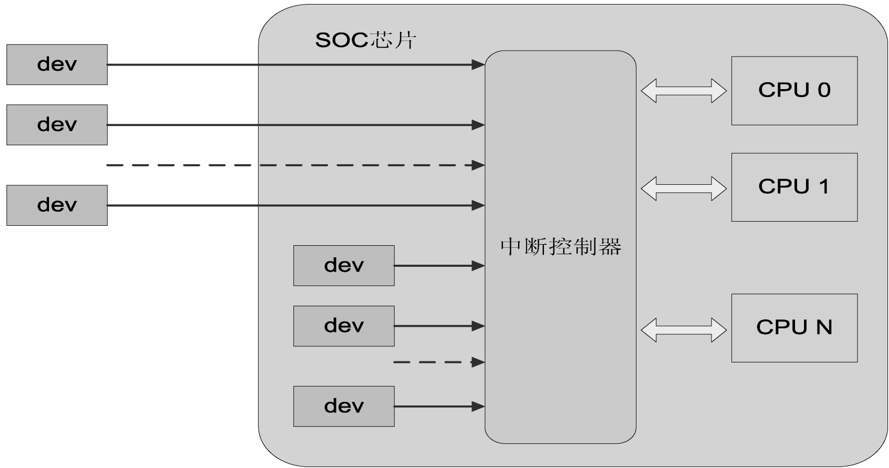


图 4-25　中断在系统中的连接示意图

从图 4-25 中可见，中断相关的硬件包括设备、中断控制器和处理器。处理器是和系统结构相关的，剩下的设备和中断控制器可能有各种不同的实现，所以中断处理框架需要能够适应各种设备和中断控制器的实现。在 SMP 的系统中，中断处理应该可以在任何或者指定的处理上执行，中断处理框架应该支持该功能。

另外，中断可以通过中断控制器进行级联，如图 4-26 所示。

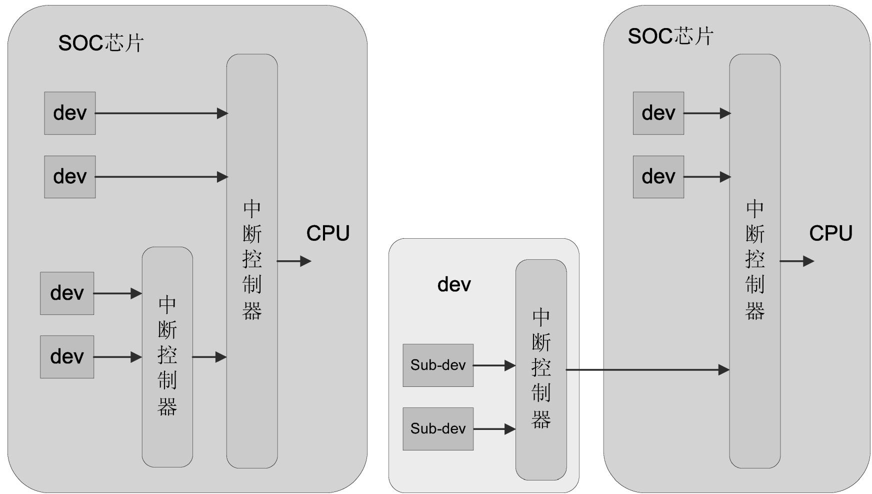


图 4-26　中断在系统中级联示意图

从图 4-26 中可见，中断控制器也可以分层次，而和 CPU 最近的中断控制器可以作为主

中断控制器。这种情况下，主中断控制器中就有某些中断只是作为级联使用，而处理器要能够区分其他中断控制器上报的中断究竟来自哪个设备，并对中断控制器进行正确的操作。除了级联之外还可能有多个设备共享同一个中断（如 PCI 总线设备），这就要求中断处理框架也要支持多个设备共享中断的功能。

从中断触发信号的角度，不能限制中断触发的形式，所以中断处理框架还需要支持不同的中断触发方式。触发信号的方式不同，带来的处理逻辑也是不同的，比如边沿触发信号就不会产生假中断的问题，而电平触发则会有假中断的问题。这需要对中断控制器进行不同的操作逻辑来避免该类问题的发生，中断处理逻辑需要和中断控制器的操作相结合以使中断处理能正确的执行。

以上主要还是功能需求，对于性能方面，中断处理都是有时间要求的，所以会有中断响应时间这一性能指标的要求。好的系统需要在大量上报中断时仍能有较小的中断响应时间。而设备的差别，造成在中断处理中需要处理的数据量是有差异的，不能因为某些设备有大量数据需要处理就影响其他设备的中断响应时间，所以在中断处理框架中也要考虑这种性能的需求。

4.3.2 中断处理框架介绍

1. 基本中断处理流程

Linux 内核的中断处理框架当然要满足之前提到的各种需求。对 Linux 内核来说，每个中断号都是通过 irq_desc 进行描述的，可以说 irq_desc 是中断处理的核心。下面来看看相应的内容：

```
struct irq_desc {
	/*
	 * This union will go away, once we fixed the direct access to
	 * irq_desc all over the place. The direct fields are a 1:1
	 * overlay of irq_data.
	 */
	//这里 union 是历史遗留问题,irq_data 和下面的 struct 内容完全相同,需要等到所有中断
	//处理代码都使用 irq_data 后,该 struct 才会被移除
	union {
		struct irq_data		irq_data;
		struct {
			unsigned int		irq;
			unsigned int		node;
			struct irq_chip		*chip;
			void			*handler_data;
			void			*chip_data;
			struct msi_desc		*msi_desc;
#ifdef CONFIG_SMP
			cpumask_var_t		affinity;
#endif
```

```
        };
    };
#endif

    struct timer_rand_state *timer_rand_state;
    unsigned int        *kstat_irqs;
    irq_flow_handler_thandle_irq;
    struct irqaction     *action;  /* IRQ action list */
    unsigned int        status;        /* IRQ status */

    unsigned int        depth;         /* nested irq disables */
    unsigned int        wake_depth;    /* nested wake enables */
    unsigned int        irq_count;     /* For detecting broken IRQs */
    unsigned long       last_unhandled; /* Aging timer for unhandled count */
    unsigned int        irqs_unhandled;
    raw_spinlock_t      lock;
#ifdef CONFIG_SMP
    const struct cpumask* affinity_hint;
#endif
    atomic_t        threads_active;
    wait_queue_head_t        wait_for_threads;
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry    *dir;
#endif
    const char          *name;
} ____cacheline_internodealigned_in_smp;
```

每个 irq_desc 是对中断号的抽象描述；而 irq_chip 是对中断控制器的抽象描述；affinity 则是对中断处理在哪些处理器上执行的抽象描述；还有对中断信号的处理方式则是通过 handle_irq 来进行描述；最后需要抽象出来的就是不同设备特别地处理了，这个工作就由 action 来完成。其他的都是表明相关操作的属性以及和中断在哪个执行实体上运行相关。通过这些抽象结构就可以实现整个中断处理的框架。当然随着 Linux 内核的演进，这些抽象数据结构的具体形式会有变化（如结构中注释所说），但是要满足中断处理的需求，框架中必然会有物理实体的抽象，这个是不会发生变化的。明白了这些抽象结构表示的物理实体，也就比较容易理解框架的执行过程和逻辑了，图 4-27 以 ARM 为例展示了中断处理代码的流程和抽象结构之间的关系。

中断处理的基本操作逻辑就是先获得正确的中断号，然后由相应中断信号处理方式 handle_irq 处理。handle_irq 会根据需要，对中断控制器进行操作，以保证中断控制器仍然能够正确地上报中断，另外根据 action 中的操作完成设备申请的操作逻辑。

2. 中断处理延时操作

下面介绍系统如何解决中断中大数据量时的延时处理问题。图 4-28 是 Linux 内核中断处理流程。

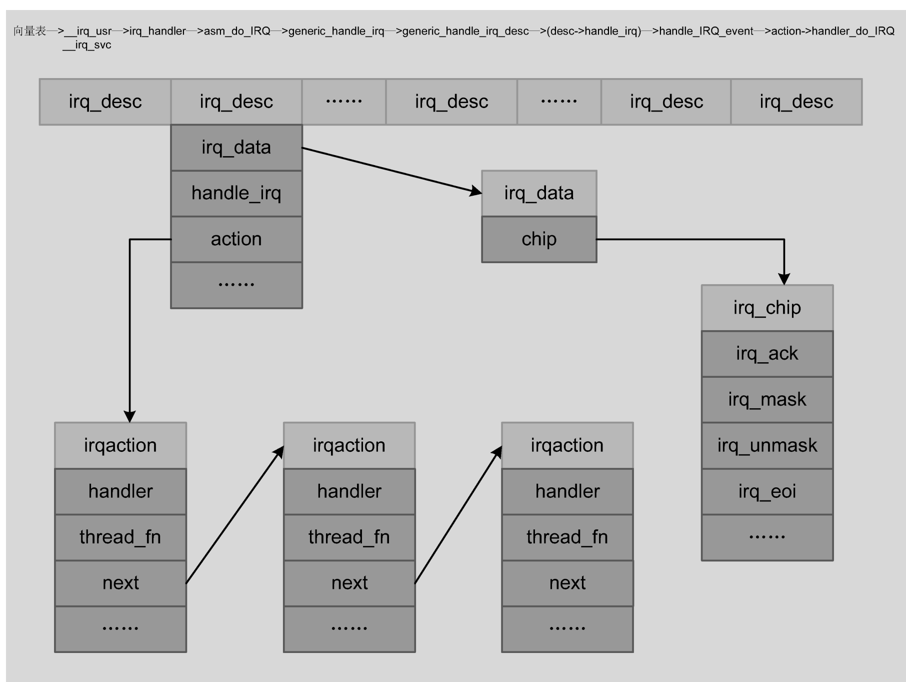


图 4-27　Linux 内核中断处理代码流程和抽象结构的关系

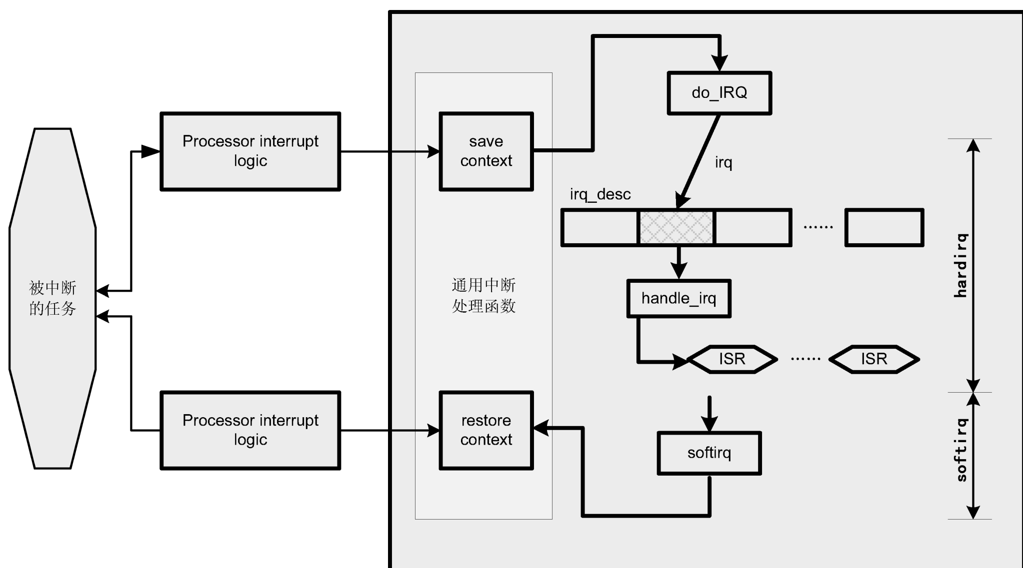


图 4-28　Linux 内核中断处理流程

从图 4-28 中可见，Linux 内核中断处理分为两部分，分别为 hardirq 和 softirq。hardirq 部分决定中断控制器是否能继续上报中断的时间，可以理解为硬件的中断延时时间，中断处理系统将需要进行大量数据处理的工作放到了 softirq 中进行。这样将完整的中断处理分为两部分进行，既保证了硬件的最小时延又保证了数据的处理。softirq 是 Linux 内核采用的比较老的机制，是对中断数据处理的一种延时操作，也可将其作为延时处理来考虑，相应的介绍有很多，这里就不进行说明了。中断的数据处理部分采用内核线程处理是一种很直接的方

式，内核也提供了该方式相应接口 request_threaded_irq。通过 ps 查看任务时，若见到如下信息（[irq/中断号 - 名字]），就说明是中断处理线程。

```
root    549  0.0  0.0    0    0 ?    S    07:20   0:00 [irq/44 - mei]
```

下面对 request_threaded_irq 接口进行一下介绍：

request_threaded_irq(unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev);

- irq——需要申请的中断号。
- handler——中断处理逻辑接口。该接口运行在中断上下文中，只是执行需要快速响应的操作,执行时间尽可能短小，耗时的工作留给线程处理接口。
- thread_fn——如果该参数不为 NULL，内核会为该 irq 创建一个内核线程。当中断发生时，如果 handler 返回值是 IRQ_WAKE_THREAD，内核将会激活中断线程，在中断线程中，thread_fn 指向的接口函数将被调用。该接口函数运行在进程上下文中，允许进行阻塞操作。
- flags——控制中断行为的位标志，形式为 IRQF_XXXX，例如：IRQF_TRIGGER_RISING、IRQF_TRIGGER_LOW、IRQF_SHARED 等。
- name——申请本中断服务的设备名称，同时也作为中断线程的名称。
- dev——作为 handler 和 thread_fn 的参数，通常设置为设备的管理实体。

3. 内核提供的通用接口

为了方便开发，内核为访问中断处理框架中抽象的控制结构提供了一些接口函数：

- irq_set_chip(irq, *chip)/irq_get_chip(irq)——每个中断号 irq_chip 的操作接口。
- irq_set_handler_data(irq, *data)/irq_get_handler_data(irq)——每个中断号内中断处理逻辑的私有数据，保存 handler 所需要的特殊数据，例如中断控制器级联时，进行级联的中断号，用该字段保存下一级中断号的管理实体。
- irq_set_chip_data(irq, *data)/irq_get_chip_data(irq)——每个中断号内操作中断控制器的私有数据，用于不同的中断控制器管理信息。
- irq_set_irq_type(irq, type)——用于设置中断的触发类型，可选的类型有 IRQ_TYPE_EDGE_RISING、IRQ_TYPE_EDGE_FALLING、IRQ_TYPE_EDGE_BOTH、IRQ_TYPE_LEVEL_HIGH、IRQ_TYPE_LEVEL_LOW。
- irq_set_handler(irq, handle)——设置中断处理逻辑，参数 handle 的类型是 irq_flow_handler_t。
- irq_set_chip_and_handler(irq, *chip, handle)——同时设置中断处理逻辑和中断控制器属性。
- irq_set_chained_handler(irq, *chip, handle)——进行级联的中断号，设置级联的中断处理逻辑，其中会设置标志 IRQ_NOREQUEST、IRQ_NOPROBE、IRQ_NOTHREAD。通常用于中断控制器的级联，设置上述三个标志位会使得父控制器的相应中断号不允许被驱动程序申请。

另外内核还提供了一些中断信号处理的方式：

- handle_level_irq。

- handle_edge_irq。
- handle_fasteoi_irq。
- handle_simple_irq。
- handle_percpu_irq。
- handle_edge_eoi_irq。
- handle_bad_irq。

为设备开发提供的中断请求的接口是 request_irq 和 request_threaded_irq。

中断子系统还提供一些接口用于动态申请/扩展中断号（需要配置 CONFIG_SPARSE_IRQ），分别如下：

- irq_alloc_desc(node)——申请一个 irq，node 是对应内存节点的编号。
- irq_alloc_desc_at(at,node)——在指定位置申请一个 irq，如果指定位置已经被占用则失败。
- irq_alloc_desc_from(from, node)——从指定位置开始搜索，申请一个 irq。
- irq_alloc_descs(irq,from,cnt,node)——申请多个连续的 irq 编号，从 from 位置开始搜索。
- irq_free_descs(irq, cnt)——释放 irq 资源。

这些只是申请函数，要想中断能够正常工作，还需要通过接口对必要的字段进行设置，如 irq_set_chip_and_handler_name、irq_set_handler_data、irq_set_chip_data。

中断处理系统还提供了接口，可以方便其他模块查询当前所在的中断处理状态，利于模块做出正确的处理。其接口如下：

- in_irq()——判断当前是否在硬件中断上下文。
- in_softirq()——判断当前是否在软件中断上下文。
- in_interrupt()——判断当前是否进行中断处理，包括在硬件、软件、底半部中断上下文。

这些接口函数是实现具体处理器及设备中断处理的基础，为各种驱动和处理器提供了良好的操作框架。

4. 电源管理相关接口

以上主要是中断处理框架的通用功能。在电源管理方面，中断还是有一定作用的。中断可以作为唤醒系统的信号，使得系统在进入待机时能够被唤醒，而这需要中断控制器的支持。为了实现该功能，系统在中断控制器的抽象实体 irq_chip 中进行了相关的设计。下面列出了 irq_chip 中的主要接口：

```
struct irq_chip {
    const char      *name;
...
    unsigned int    (*irq_startup)(struct irq_data *data);
    void        (*irq_shutdown)(struct irq_data *data);
    void        (*irq_enable)(struct irq_data *data);
    void        (*irq_disable)(struct irq_data *data);

    void        (*irq_ack)(struct irq_data *data);
    void        (*irq_mask)(struct irq_data *data);
    void        (*irq_mask_ack)(struct irq_data *data);
```

```
    void      ( * irq_unmask)(struct irq_data  * data);
...
    int       ( * irq_set_wake)(struct irq_data  * data, unsigned int on);
...
};
```

其中，irq_set_wake 是和唤醒相关的功能，主要是打开或关闭相应中断唤醒系统的功能。

内核提供对某个中断进行电源管理唤醒功能的操作接口，接口为 int set_irq_wake(unsigned int irq, unsigned int on);

可见内核中断处理框架提供了包括功能、性能和电源管理相关的完整系统方案。

4.3.3 TI 芯片中断处理相关实现详解

首先来看看 DM 3730 中断控制器的框架，如图 4-29 所示。图 4-29 引自《DM 3730 芯片手册》中第 2406 页的框图。

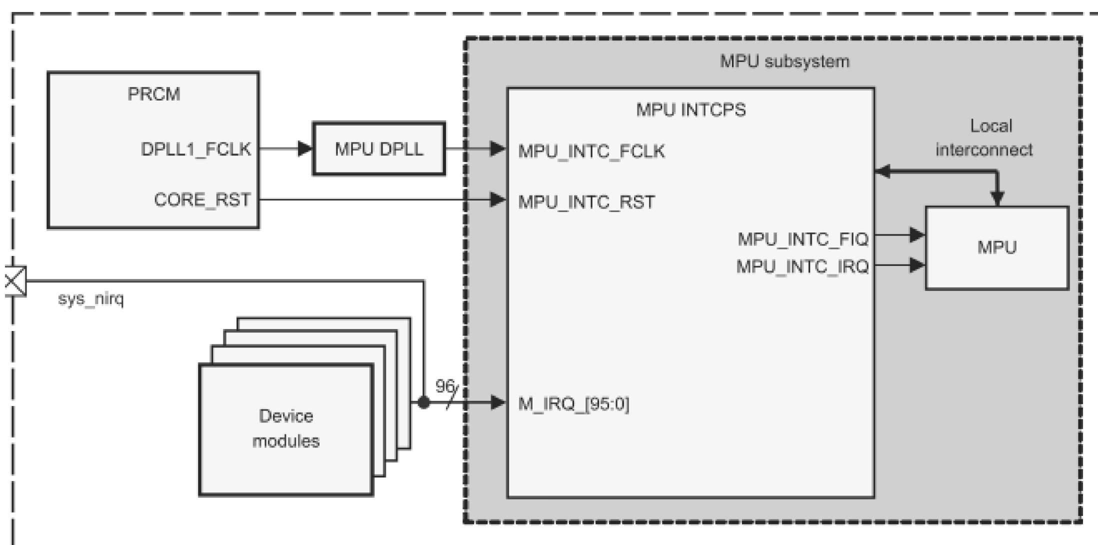


图 4-29　DM 3730 中断控制器框图

从图 4-29 可见，中断控制器的时钟等信号是在 PRCM 的管理之下，所以相应的也有电源管理的功能，后续的代码注释中对相关的代码进行了说明。但是由于电源管理的框架还没有进行介绍，所以中断控制器中相关电源管理的代码可以先浏览，等到了解电源管理框架之后会有更加深入的理解。

接下来分步说明中断处理的具体实现。

1. 中断号实现

从图 4-29 中可见，中断控制器有 96 个中断，这些中断号是如何定义的，又如何让系统知道正确的中断数目的呢？

Linux 内核 irqdesc. h 中有如下定义：

```
extern struct irq_desc irq_desc[NR_IRQS];
```

这里可见宏 NR_IRQS 描述了所有中断的个数，这个数目应该是和处理器相关的。DM 3730 上 NR_IRQS 的具体值以及中断号的定义在 arch/arm/plat-omap/include/mach/irqs.h 中可见，具体如下：

```
#define INT_34XX_BENCH_MPU_EMUL   3
define INT_34XX_ST_MCBSP2_IRQ   4
#define INT_34XX_ST_MCBSP3_IRQ   5
…
#define INT_34XX_GPIO_BANK1   29
#define INT_34XX_GPIO_BANK2   30
#define INT_34XX_GPIO_BANK3   31
#define INT_34XX_GPIO_BANK4   32
#define INT_34XX_GPIO_BANK5   33
#define INT_34XX_GPIO_BANK6   34
…
#define INT_34XX_MMC3_IRQ   94
#define INT_34XX_GPT12_IRQ   95
…
#define NR_IRQS                    OMAP_GPMC_IRQ_END
```

这里为什么不是 96 的数值，而是另外的一个宏呢？其实这是为了解决中断级联的问题。之前在框架设计中已经看到，通过 irq_action 可以解决中断共享，但是并没有说明如何解决中断级联。其实中断级联就是通过 irq_desc 自身解决的，比如 GPIO 做中断源的话就会产生级联中断的情况，这样中断号就需要增加，就像需求中讨论的某些中断号（如 INT_34XX_GPIO_BANKx）会作为特殊中断来进行处理，在 Linux 内核中，是通过 set_irq_chained_handler 来实现该功能的。注意 DM 3730 内核不支持配置 CONFIG_SPARSE_IRQ，所以不能用动态申请的方式支持级联，而只能采用静态扩大中断号的方式。具体的级联中断的解决办法留到相应模块中进行详细说明。

2. ARM 向量表到中断处理的实现

ARM 内核的中断处理是从向量表开始的。向量表在 arch/arm/kernel/entry_armv.S 中定义：

```
    .globl   __stubs_start
__stubs_start:

    vector_stub irq, IRQ_MODE, 4

    .long    __irq_usr            @  0  (USR_26/USR_32)
    .long    __irq_invalid            @  1  (FIQ_26/FIQ_32)
```

```
        .long   __irq_invalid           @  2  (IRQ_26/IRQ_32)
        .long   __irq_svc               @  3  (SVC_26/SVC_32)

        vector_stub dabt, ABT_MODE, 8

        .long   __dabt_usr              @  0  (USR_26/USR_32)
        .long   __dabt_invalid          @  1  (FIQ_26/FIQ_32)
        .long   __dabt_invalid          @  2  (IRQ_26/IRQ_32)
        .long   __dabt_svc              @  3  (SVC_26/SVC_32)

vector_fiq:
        disable_fiq
        subs    pc, lr, #4
        ……
        .globl  __stubs_end
__stubs_end:

        .equ    stubs_offset, __vectors_start + 0x200 - __stubs_start
        .globl  __vectors_start
__vectors_start:
ARM(    swi SYS_ERROR0  )
THUMB(  svc #0          )
THUMB(  nop             )
        W(b)    vector_und + stubs_offset
        W(ldr)  pc, .LCvswi + stubs_offset
        W(b)    vector_pabt + stubs_offset
        W(b)    vector_dabt + stubs_offset
        W(b)    vector_addrexcptn + stubs_offset
        W(b)    vector_irq + stubs_offset
        W(b)    vector_fiq + stubs_offset

        .globl  __vectors_end
__vectors_end:
```

这部分代码主要分为两部分：其一是真正的向量表，位于__vectors_start 和__vectors_end 之间；其二是处理跳转的部分，位于__stubs_start 和__stubs_end 之间。

```
vector_stub irq, IRQ_MODE, 4
```

这句其实是宏把宏展开后就定义了 vector_irq。根据进入中断前的处理器模式，分别跳转到__irq_usr 或__irq_svc。

```
vector_stub dabt, ABT_MODE, 8
```

这句宏展开后就定义了 vector_dabt，根据进入中断前的处理器模式，分别跳转到__ dabt_usr 或__ dabt_svc。

在系统启动阶段，由 early_trap_init( )（位于 arch/arm/kernel/traps. c 中）初始化向量表的过程如下：

```
void __ init early_trap_init(void)
{
    ……
    /*
     * Copy the vectors, stubs and kuser helpers(in entry - armv. S)
     * into the vector page, mapped at 0xffff0000, and ensure these
     * are visible to the instruction stream.
     */
    memcpy((void *)vectors, __ vectors_start, __ vectors_end - __ vectors_start);
    memcpy((void *)vectors +0x200, __ stubs_start, __ stubs_end - __ stubs_start);
    ……
}
```

以上两个 memcpy 会把__vectors_start 开始的代码复制到 0xffff0000 处，把__stubs_start 开始的代码复制到 0xffff0000 +0x200 处，和相对跳转的偏移相同。这样异常中断到来时，CPU 就可以正确地跳转到相应中断向量入口并执行了。

为什么要将__ stubs_start 开始的内容进行如此的复制呢？这里说明一下，先看看内核编译后的符号表：

```
c000b4a0 T __ stubs_start
c000b4a0 t vector_irq
c000b520 t vector_dabt
c000b5a0 t vector_pabt
c000b620 t vector_und
c000b6a0 t vector_fiq
c000b6a4 t vector_addrexcptn
c000b6c4 T __ stubs_end
c000b6c4 T __ vectors_start
c000b6e4 T __ vectors_end
```

可见__ stubs_start 在向量表内容之前，而根据向量处理的映射，整个向量处理的代码都需要在 0xffff0000 之后的一个页内，所以就需要将__ stubs_start 开始的内容进行复制，这也是需要进行偏移修正的原因。

对于中断使用 vector_stub 方式如下：

```
        vector_stub     irq, IRQ_MODE, 4
```

进一步分析 vector_stub：

```
        .macro  vector_stub, name, mode, correction=0
        .align  5
vector_\name:
        .if \correction
        sub lr, lr, #\correction   @异常时硬件会将 lr 加偏移,要修正才能得到正确的返回值
        .endif
        @ Save r0, lr_<exception>(parent PC) and spsr_<exception>
        @ (parent CPSR)
        stmia   sp, {r0, lr}            @ save r0, lr
        mrs lr, spsr                    @lr 保存 spsr 的值为后续使用,spsr 为进入向量之前的状态
        str lr, [sp, #8]                @ save spsr

        @ Prepare for SVC32 mode.  IRQs remain disabled.
        mrs r0, cpsr
        eor r0, r0, #(\mode ^ SVC_MODE | PSR_ISETSTATE)
        msr spsr_cxsf, r0   @把 r0 的值写入 spsr 寄存器(cxsf 表示要写哪些位)
                            @此时 spsr 的值已经包含了 SVC_MODE
        @ the branch table must immediately follow this code
        and lr, lr, #0x0f   @lr 为保存的 spsr 的值,此语句取到进入异常前的模式
THUMB(  adr r0, 1f                  )
THUMB(  ldr lr, [r0, lr, lsl #2]    )
        mov r0, sp
ARM(    ldr lr, [pc, lr, lsl #2]    )       @lr=pc+mode×4 根据模式获得正确的指令地址
                                            @根据 thumb 指令可知 pc 指针值为 1 标号处,
                                            @pc 为此值和指令流水有关
        movs    pc, lr                  @ branch to handler in SVC mode
                                        @这里会将 spsr 写入 cpsr 保证是在 SVC 模式执行处理
ENDPROC(vector_\name)

        .align  2
        @ handler addresses follow this label
1:
        .endm
```

中断处理上述代码，最终会根据之前的模式进入__irq_usr 或者__irq_svc 执行，二者最终都会进入 irq_handler 这个宏：

```
        . macro irq_handler
        get_irqnr_preamble r5, lr
1:      get_irqnr_and_base r0, r6, r5, lr
        movne   r1, sp
        @
        @ routine called with r0 = irq number, r1 = struct pt_regs *
        @
        adrne       lr, BSYM(1b)
        bneasm_do_IRQ
```

get_irqnr_preamble 和 get_irqnr_and_base 这两个宏由板级支持的代码在 arch/arm/plat-omap/include/mach/entry-macro.S 中定义，目的就是从中断控制器中获得正确的中断号，来看看相应的实现：

```
#if defined(CONFIG_ARCH_OMAP2) || defined(CONFIG_ARCH_OMAP3)
        . macro get_irqnr_preamble, base, tmp
#ifdef CONFIG_ARCH_OMAP2
        ldr   \base, =OMAP2_IRQ_BASE
#else
        ldr   \base, =OMAP3_IRQ_BASE
#endif
        . endm

        /* Check the pending interrupts. Note that base already set */
        . macro   get_irqnr_and_base, irqnr, irqstat, base, tmp
        ldr   \irqnr, [\base, #0x98] /* IRQ pending reg 1 */
        cmp \irqnr, #0x0
        bne9999f
        ldr\irqnr, [\base, #0xb8] /* IRQ pending reg 2 */
        cmp\irqnr, #0x0
        bne9999f
        ldr\irqnr, [\base, #0xd8] /* IRQ pending reg 3 */
        cmp\irqnr, #0x0
9999:
        ldrne   \irqnr, [\base, #INTCPS_SIR_IRQ_OFFSET]
        and\irqnr, \irqnr, #ACTIVEIRQ_MASK /* Clear spurious bits */
        . endm
#endif

/*
 * Optimized irq functions for ti81xx
 */
```

```
#ifdef CONFIG_ARCH_TI81XX
        .macro get_irqnr_preamble, base, tmp
        ldr\base, =OMAP3_IRQ_BASE
        .endm
        /* Check the pending interrupts. Note that base already set */
        .macroget_irqnr_and_base, irqnr, irqstat, base, tmp
        ldr\irqnr, [\base, #0x98] /* IRQ pending reg 1 */
        cmp\irqnr, #0x0
        bne9999f
        ldr\irqnr, [\base, #0xb8] /* IRQ pending reg 2 */
        cmp\irqnr, #0x0
        bne9999f
        ldr\irqnr, [\base, #0xd8] /* IRQ pending reg 3 */
        cmp\irqnr, #0x0
        bne9999f
        ldr\irqnr, [\base, #0xf8] /* IRQ pending reg 4 */
        cmp\irqnr, #0x0
9999:
        ldrne   \irqnr, [\base, #INTCPS_SIR_IRQ_OFFSET]
        and\irqnr, \irqnr, #ACTIVEIRQ_MASK
        .endm
#endif

#ifdef CONFIG_ARCH_OMAP4
        .macro get_irqnr_preamble, base, tmp
        ldr   \base, =OMAP4_IRQ_BASE
        .endm
        /*
         * The interrupt numbering scheme is defined in the
         * interrupt controller spec. To wit:
         *
         * Interrupts 0-15 are IPI
         * 16-28 are reserved
         * 29-31 are local. We allow 30 to be used for the watchdog.
         * 32-1020 are global
         * 1021-1022 are reserved
         * 1023 is "spurious" (no interrupt)
         *
         * For now, we ignore all local interrupts so only return an
         * interrupt if it's between 30 and 1020. The test_for_ipi
         * routine below will pick up on IPIs.
```

```
    * A simple read from the controller will tell us the number
    * of the highest priority enabled interrupt.
    * We then just need to check whether it is in the
    * valid range for an IRQ(30 - 1020 inclusive).
    */
   .macro get_irqnr_and_base, irqnr, irqstat, base, tmp
   ldr \irqstat, [\base, #GIC_CPU_INTACK]
   ldr \tmp, =1021
   bic \irqnr, \irqstat, #0x1c00
   cmp \irqnr, #29
   cmpcc \irqnr, \irqnr
   cmpne \irqnr, \tmp
   cmpcs \irqnr, \irqnr
   .endm
#endif
```

从这可见，无论 DM 3730、TI 816X 还是 OMAP4 都有自己的实现，其中比较重要的是宏 OMAPx_IRQ_BASE，这是中断控制器的基地址，get_irqnr_preamble 就是通过该宏获得中断控制器的基地址的。而 get_irqnr_and_base 也需要该基地址来获得正确的触发中断的中断号。在 irq_handler 获得正确的中断号之后，紧接着就调用 asm_do_IRQ，从这个函数开始，中断程序进入 C 代码中。该函数在 arch/arm/kernel/irq.c 中，实现如下：

```
asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);

    irq_enter();

    /*
     * Some hardware gives randomly wrong interrupts.  Rather
     * than crashing, do something sensible.
     */
    if(unlikely(irq >= nr_irqs)) {
        if(printk_ratelimit())
            printk(KERN_WARNING "Bad IRQ%u\n", irq);
        ack_bad_irq(irq);
    } else {
        generic_handle_irq(irq);
    }

    /* AT91 specific workaround */
    irq_finish(irq);
    irq_exit();
    set_irq_regs(old_regs);
}
```

到这里，中断处理就完成了从向量表到体系结构无关的 generic_handle_irq 中断处理的转换，后续可以根据正确的中断号进行中断处理了。

3. 中断控制器的处理

接下来看看 DM 3730 的内核是如何实现中断控制器的管理的。首先，看一下 DM 3730 中断控制器的内部框架，如图 4-30 所示。图 4-30 引自《DM 3730 芯片手册》中第 2411 页的框图。

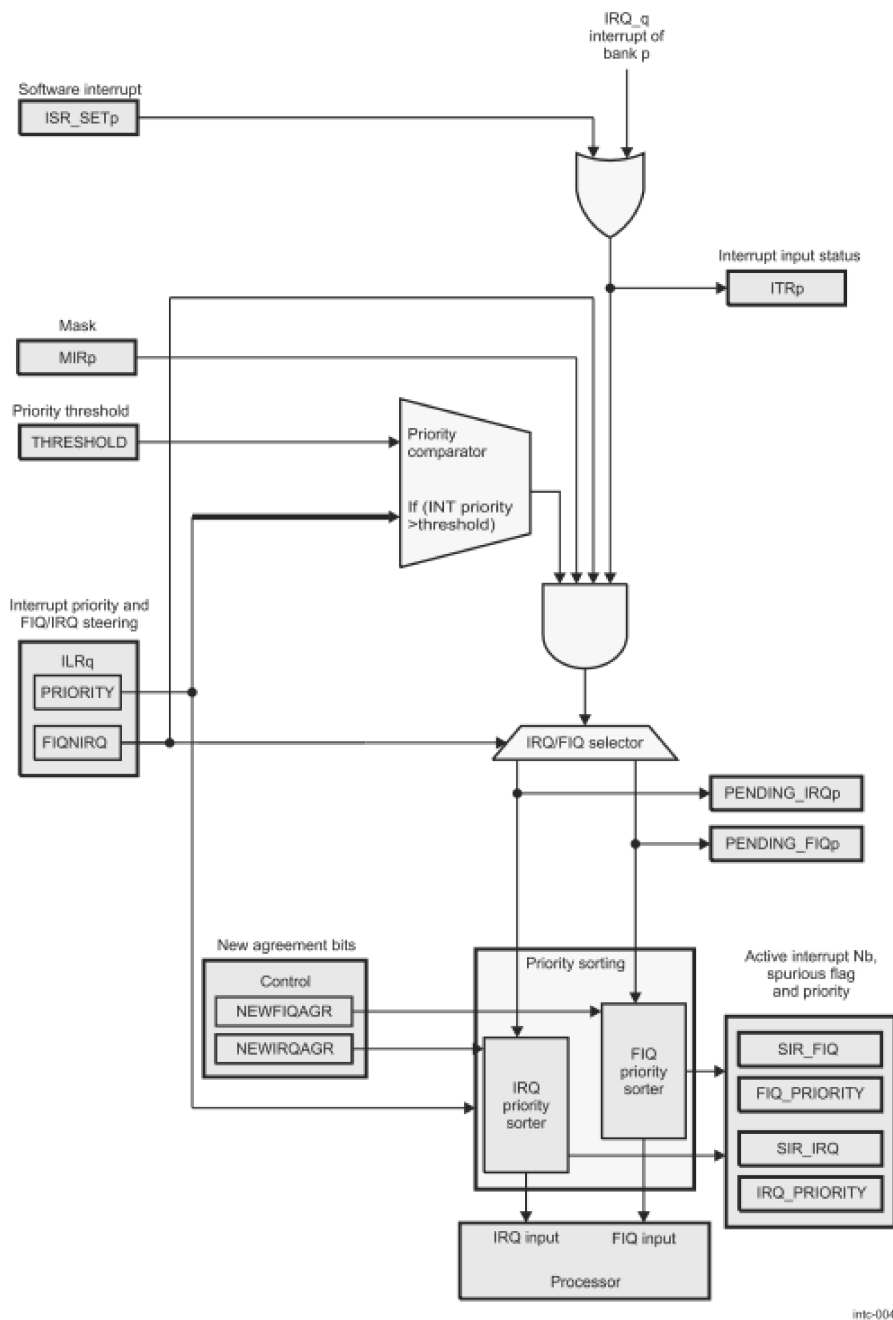


图 4-30　DM 3730 中断控制器内部框架

从图 4-30 中可见，该中断控制器可进行中断的优先级管理，并能屏蔽中断，以及伪中断的检查，这些都是内核需要的中断控制器的功能。DM 3730 中断控制器的实现代码在 linux/arch/arm/mach-omap2/irq.c 中。中断控制器的管理主要是实现 irq_chip 的操作接口，该接口提供给内核的中断处理框架，通过这些接口，内核可在其提供的标准中断处理流程接口 irq_flow_handler_t 中调用如 ack、mask、unmask 等操作，从而正确完成整个的中断处理流程。现将中断控制器管理相关的代码整理到一起，并注释如下：

```
/* selected INTC register offsets */
//中断控制器寄存器偏移地址
#define INTC_REVISION           0x0000
#define INTC_SYSCONFIG          0x0010
#define INTC_SYSSTATUS          0x0014
#define INTC_SIR            0x0040
#define INTC_CONTROL            0x0048
#define INTC_PROTECTION             0x004C
#define INTC_IDLE0          x0050
#define INTC_THRESHOLD0             x0068
#define INTC_MIR00          x0084
#define INTC_MIR_CLEAR00            x0088
#define INTC_MIR_SET00          x008c
#define INTC_PENDING_IRQ00          x0098
/* Number of IRQ state bits in each MIR register */
#define IRQ_BITS_PER_REG      32

//该结构是针对如果 omap 中包含多个 intc 控制 bank,则每个 bank 有自己的相应管理实体。为扩
//展留有余地
static struct omap_irq_bank {
    void __iomem *base_reg;
    unsigned int nr_irqs;
} __attribute__((aligned(4)))irq_banks[] = {
    {
        /* MPU INTC */
        .nr_irqs = 96,
    },
};
//中断控制器寄存器的操作接口函数,其中均为虚拟地址。对于 DM 3730 相关的管理实体就是
//irq_banks[0]
static void intc_bank_write_reg(u32 val, struct omap_irq_bank *bank, u16 reg)
{
    __raw_writel(val, bank->base_reg + reg);
}
```

```
    static u32 intc_bank_read_reg(struct omap_irq_bank *bank, u16 reg)
{
    return __raw_readl(bank->base_reg + reg);
}

    static int previous_irq;

/*
 * On 34xx we can get occasional spurious interrupts if the ack from
 * an interrupt handler does not get posted before we unmask. Warn about
 * the interrupt handlers that need to flush posted writes.
 */
//检查伪中断。由于 DM 3730 的中断控制器是电平触发的,如果中断处理程序处理不当,很可能
//会有中断误报的伪中断现象,这里可进行检查并报错。使用该函数主要是为了发现中断处理函
//数的问题
//例如出现 warning Spurious irq 95: 0xffffffdf, please flush posted write for irq 86

//通常设备的中断处理函数中,需要写相应的寄存器,表明中断处理了,这样相应的设备模块可以
//停止给中断控制器发送中断请求。由于对于寄存器的写操作是异步,如果相应的寄存器还没有
//生效,而由于内核的中断处理函数在调用 irq_handler 之后会调用中断控制器的 umask,此时如
//果正有中断报给中断控制器,在硬件上就会造成伪中断 spurious 的问题
//这时需要将相应中断号的设备中断处理函数的寄存器读一下以保证写完成

//《DM 3730 芯片手册》中具体的 spurious 说明(在 2421 页)如下
// The spurious flag indicates whether the result of the sorting(a window of 10 INTC
// functional clock cycles after the interrupt assertion) is invalid. The sorting is invalid if:
// The interrupt that triggered the sorting is no longer active during the sorting.
// A change in the mask has affected the result during the sorting time.
static int omap_check_spurious(unsigned int irq)
{
    u32 sir, spurious;

    //就是通过读 INTC_SIR 的高位,其中包含伪中断标识符
    sir = intc_bank_read_reg(&irq_banks[0], INTC_SIR);
    spurious = sir >> 7;

    if(spurious) {
        printk(KERN_WARNING "Spurious irq %i: 0x%08x, please flush "
                    "posted write for irq %i\n",
                    irq, sir, previous_irq);
        return spurious;
    }
```

```
    return 0;
}

/* XXX: FIQ and additional INTC support(only MPU at the moment) */
//DM 3730 的 irq ack 函数主要是为中断控制器的 ack 函数接口
static void omap_ack_irq(unsigned int irq)
{
    //操作 INTCPS_CONTROL 寄存器,往该寄存器的 NEWIRQAGR 位写 1
    //表明重置中断输出,使能新的中断源
    intc_bank_write_reg(0x1, &irq_banks[0], INTC_CONTROL);
}

//屏蔽指定的 irq,在该函数中要做可能产生 spurious 中断的记录.
static void omap_mask_irq(unsigned int irq)
{
    //该 offset 为相应的 MIR_SETn 寄存器的 n 值,表示第 n 个寄存器
    //根据 irq 号进行下面计算即可
    int offset = irq &(~(IRQ_BITS_PER_REG - 1));

    if(cpu_is_omap34xx()){
        int spurious = 0;

        /*
         * INT_34XX_GPT12_IRQ is also the spurious irq. Maybe because
         * it is the highest irq number?
         */
        //这里通过定时中断来定期检查是否有 spurious 伪中断产生,
        //如果有则显示信息
        if(irq == INT_34XX_GPT12_IRQ)
            spurious = omap_check_spurious(irq);

        //这里记录相应的中断 previous_irq 是为了记录可能的 spurious 伪中断。由于这里是
        //mask irq,而之前分析了可能产生 spurious 伪中断是之前的 umask irq,所以这里记
        //录相应的 irq 号到 previous_irq
        //而下一次如果 omap_check_spurious 中返回非 0,表明有 spurious 则说明此次记录到
        //previous_irq 的 irq 就是相应的 spurious irq
        if(! spurious)
            previous_irq = irq;
    }

    //有了相应的 offset,那么寄存器中应该写入低 32bits 的值
```

```
        //通过与操作只保留低 32bits 值
        irq &= (IRQ_BITS_PER_REG - 1);

        //通过 intc 接口写入相应的寄存器,omap3 只有 bank0
        intc_bank_write_reg(1 << irq, &irq_banks[0], INTC_MIR_SET0 + offset);
}

//irq 的 umask 操作,通过写 MIR_CLEAR 寄存器实现
static void omap_unmask_irq(unsigned int irq)
{
        //计算 offset,同 mask 操作
        int offset = irq &(~(IRQ_BITS_PER_REG - 1));

        //保留低 32bits 并写寄存器
        irq &= (IRQ_BITS_PER_REG - 1);

        intc_bank_write_reg(1 << irq, &irq_banks[0], INTC_MIR_CLEAR0 + offset);
}

//该函数作为 irq ack 的接口函数,ARM 核本身有 MPU_INTC_FIQ 和 MPU_INTC_IRQ 两个
//中断请求信号
static void omap_mask_ack_irq(unsigned int irq)
{
        //首先要 mask 相应的 irq,避免外部模块的 irq 重复报告给 ARM
        //由于这里是在 ARM cortex A8 core 之外的 intc,需要先 mask irq
        omap_mask_irq(irq);

        //该 ack 实现写 intcps_control 寄存器来重新使能 intc_irq
        omap_ack_irq(irq);
}

//这里是 ARM irq 的重要的中断控制器的接口 irq_chip
static struct irq_chip omap_irq_chip = {
        .name = "INTC",
        .ack = omap_mask_ack_irq,
        .mask = omap_mask_irq,
        .unmask = omap_unmask_irq,
};
```

4. 中断控制器初始化

中断控制器初始化相关的代码和具体实现说明如下:

```
//对相应 bank 的 intc 进行初始化,主要是 soft reset 并设置内部时钟为 autoidle(节能)
```

```
static void __ init omap_irq_bank_init_one(struct omap_irq_bank *bank)
{
    unsigned long tmp;

    //打印 intc 的版本信息
    tmp = intc_bank_read_reg(bank, INTC_REVISION) & 0xff;
    printk(KERN_INFO "IRQ: Found an INTC at 0x%p "
             "(revision %ld.%ld) with %d interrupts\n",
             bank -> base_reg, tmp >> 4, tmp & 0xf, bank -> nr_irqs);

    //通过 sysconfig 寄存器 reset 中断控制器
    tmp = intc_bank_read_reg(bank, INTC_SYSCONFIG);
    tmp |= 1 << 1;/* soft reset */
    intc_bank_write_reg(tmp, bank, INTC_SYSCONFIG);

    //通过读取 sysstatus 寄存器确认 reset 成功结束
    while(!(intc_bank_read_reg(bank, INTC_SYSSTATUS) & 0x1))
        /* Wait for reset to complete */;

    //enable 内部 clock 的 auto idle 模式
    /* Enable autoidle */
    intc_bank_write_reg(1 << 0, bank, INTC_SYSCONFIG);
}

//内核中板级的中断初始化接口。在 machine_desc 中的 init_irq 会调用该接口, 该函数记录相应
//的中断控制器的地址并作 ioremap 供内核使用,还对中断控制器进行初始化等操作,最后最重要
//的是要对内核的中断描述符 irq_desc 进行必要的初始化
void __ init omap_init_irq(void)
{
    unsigned long nr_of_irqs = 0;
    unsigned int nr_banks = 0;
    int i;

    //对于每个硬件的 intc 模块都要初始化
    //在 DM 3730 中只有一个 intc 供 ARM 使用,所以实际只初始化一次
    for(i = 0; i < ARRAY_SIZE(irq_banks); i ++ ){
        unsigned long base = 0;
        struct omap_irq_bank *bank = irq_banks + i;

        //先记录实际的 intc 寄存器的物理基地址
        if(cpu_is_omap24xx())
            base = OMAP24XX_IC_BASE;
```

```
        else if(cpu_is_omap34xx())
            base = OMAP34XX_IC_BASE;
        else if(cpu_is_ti81xx()){
            base = TI81XX_ARM_INTC_BASE;
            bank->nr_irqs = 128;
        }

        //基地址为0,说明有bug,要报告
        BUG_ON(!base);

        /* Static mapping, never released */
        //映射中断控制器基地址
        bank->base_reg = ioremap(base, SZ_4K);
        if(!bank->base_reg){
            printk(KERN_ERR "Could not ioremap irq bank%i\n", i);
            continue;
        }

        //对 intc 进行初始化,包括控制器的 soft reset 等操作
        omap_irq_bank_init_one(bank);

        //记录到目前位置初始化后有多少中断可以激活
        nr_of_irqs += bank->nr_irqs;
        nr_banks++;
    }

    printk(KERN_INFO "Total of %ld interrupts on %d active controller%s\n",
        nr_of_irqs, nr_banks, nr_banks > 1 ? "s" : "");

    //对内核的中断描述符进行初始化,要正确的设置中断控制器管理的 irq_desc 的中断控制
    //器属性。0 - nr_of_irqs 为中断控制管理的中断号
    for(i = 0; i < nr_of_irqs; i++){
        //设置中断控制器的描述符,主要是对中断控制器的操作接口
        set_irq_chip(i, &omap_irq_chip);
        //TRM 中描述了 intc 只支持 level 触发的中断,这里设置相应的管理中断
        //handle 为 kernel 标准的 level 触发中断处理函数
        set_irq_handler(i, handle_level_irq);
        //设置 valid 标识,实际表明该中断号可以 request 了.
        set_irq_flags(i, IRQF_VALID);
    }
}
```

5. 电源管理相关

最后来看看电源管理相关的代码，先对模块的电源管理有个初步的了解。

```
/* Structure to save interrupt controller context */
//该结构是针对 power 相关的 save 和 restore 在 memory 中保存的寄存器实体,
//以便 wakeup reset 的时候可以恢复
struct omap3_intc_regs {
	u32 sysconfig;
	u32 protection;
	u32 idle;
	u32 threshold;
	u32 ilr[INTCPS_NR_IRQS];
	u32 mir[INTCPS_NR_MIR_REGS];
};

//该函数是留给 power management 的接口,在进入 sleep 之前进行最后的检查,看是否有 pending
//的中断需要处理。进入睡眠的操作过程中是不能发生中断从而引起上下文切换的,所以在准备
//进入 sleep 的最后阶段首先要 disable 中断,而在跳到 sram 的 sleep 执行之前需要检查是否有
//pending 的中断,如果有则应该直接恢复并 enable 中断以执行中断处理。这里就是进行中断
//pending 的检查
//通常该函数会在 cpu idle 的接口中被调用
int omap_irq_pending(void)
{
	int i;

	for(i = 0; i < ARRAY_SIZE(irq_banks); i++){
		struct omap_irq_bank *bank = irq_banks + i;
		int irq;

		for(irq = 0; irq < bank->nr_irqs; irq += 32)
			if(intc_bank_read_reg(bank, INTC_PENDING_IRQ0 +
						((irq >> 5) << 5)))
				return 1;
	}
	return 0;
}

#ifdef CONFIG_ARCH_OMAP3
//这部分是 DM 3730 所需要的.
//该 intc_context 为 power 相关的、需要做寄存器保护的上下文.
static struct omap3_intc_regs intc_context[ARRAY_SIZE(irq_banks)];

//保存相应的 intc 的上下文,作为 core_save_context 的一部分.
```

```
void omap_intc_save_context(void)
{
    int ind = 0, i = 0;
    for(ind = 0; ind < ARRAY_SIZE(irq_banks); ind ++){
        struct omap_irq_bank  * bank = irq_banks + ind;
        intc_context[ind]. sysconfig =
            intc_bank_read_reg(bank, INTC_SYSCONFIG);
        intc_context[ind]. protection =
            intc_bank_read_reg(bank, INTC_PROTECTION);
        intc_context[ind]. idle =
            intc_bank_read_reg(bank, INTC_IDLE);
        intc_context[ind]. threshold =
            intc_bank_read_reg(bank, INTC_THRESHOLD);
        for(i = 0; i < INTCPS_NR_IRQS; i ++)
            intc_context[ind]. ilr[i] =
                intc_bank_read_reg(bank,(0x100 + 0x4 * i));
        for(i = 0; i < INTCPS_NR_MIR_REGS; i ++)
            intc_context[ind]. mir[i] =
                intc_bank_read_reg(&irq_banks[0], INTC_MIR0 +
                (0x20  *  i));
    }
}

//intc 的上下文恢复,同样作为 core_restore_context 的一部分
void omap_intc_restore_context(void)
{
    int ind = 0, i = 0;

    for(ind = 0; ind < ARRAY_SIZE(irq_banks); ind ++){
        struct omap_irq_bank  * bank = irq_banks + ind;
        intc_bank_write_reg(intc_context[ind]. sysconfig,
                        bank, INTC_SYSCONFIG);
        intc_bank_write_reg(intc_context[ind]. sysconfig,
                        bank, INTC_SYSCONFIG);
        intc_bank_write_reg(intc_context[ind]. protection,
                        bank, INTC_PROTECTION);
        intc_bank_write_reg(intc_context[ind]. idle,
                        bank, INTC_IDLE);
        intc_bank_write_reg(intc_context[ind]. threshold,
                        bank, INTC_THRESHOLD);
        for(i = 0; i < INTCPS_NR_IRQS; i ++)
            intc_bank_write_reg(intc_context[ind]. ilr[i],
```

```
                    bank,(0x100 +0x4 * i));
            for(i =0; i < INTCPS_NR_MIR_REGS; i ++)
                intc_bank_write_reg(intc_context[ind]. mir[i],
                    &irq_banks[0], INTC_MIR0 +(0x20  *  i));
        }
        /* MIRs are saved and restore with other PRCM registers */
    }

    //清除 pending 的 irq,这样可以使系统进入 sleep 模式。在系统 suspend 需要强制进入 sleep 时使用
    void omap3_intc_suspend(void)
    {
        /* A pending interrupt would prevent OMAP from entering suspend */
        //由于 pending 的中断会阻止系统进入 suspend 模式,通过 ack 操作可以将 pending 的中断
        //清除,接收新的中断从而保证系统可以进入 suspend 模式.
        omap_ack_irq(0);
    }

    //进入 idle 的准备操作,该操作在 sram_idle 中被调用.
    void omap3_intc_prepare_idle(void)
    {
        /*
         * Disable autoidle as it can stall interrupt controller,
         * cf. errata ID i540 for 3430(all revisions up to 3. 1. x)
         */
        //该操作为勘误中的说明,由于内部 ocp clock 在 auto idle 情况下会自动 gating,而此情况下系
        //统唤醒时,由于 intc 没有内部的 clock 会导致无法产生对 ARM 的中断,从而无法唤醒整个系统
        //workaround 就是在 sleep 之前将 intc 的 autoidle 关掉,保证系统唤醒
        //时 intc 的内部 clock 不会 gating
        intc_bank_write_reg(0, &irq_banks[0], INTC_SYSCONFIG);
    }

    //intc 唤醒后的恢复操作.
    void omap3_intc_resume_idle(void)
    {
        /* Re - enable autoidle */
        //为了省电,重新 enable intc 的 autoidle,从而可以内部 clock auto gating.
        intc_bank_write_reg(1, &irq_banks[0], INTC_SYSCONFIG);
    }
    #endif/* CONFIG_ARCH_OMAP3 */
```

电源管理没有中断控制器唤醒的相关实现，由于这部分是在 ARM 主处理器的电源域中，并不需要这部分功能。会由其他的部分来唤醒系统。

4.4 内存管理

4.4.1 内存管理的基本需求

内存管理是 Linux 内核非常重要的功能，可以说是最基础的功能之一。在介绍最小系统的时候，就已经看到内存是硬件必不可少的部分。如何管理内存一直是一个综合的问题，对于内存的管理是多方面的，因为存储系统本身就是一个有层次概念的系统。存储系统的层次结构如图 4-31 所示。而内存 RAM 是既可以用来执行指令又可以用来存放数据的多功能存储器件，而且在存储系统层次中位于中间的位置，起到承上启下的作用，性价比（速度快并且价格低）十分高，这样就附加了内存缓存的功能需求，用于缓存底层存储系统的数据。

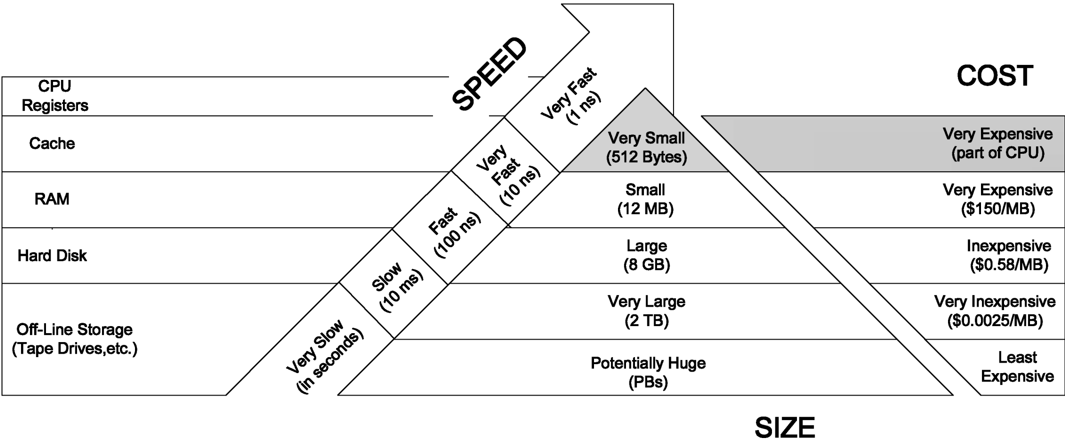


图 4-31　存储系统层次结构

整个内存管理的首要需求自然是内存物理空间的管理。当然空间的管理要有单元的概念，并且单元大小要合适，单元小则需要的管理资源多，单元大就会有浪费。由于处理器是使用自己的视角（地址空间）访问内存，内存的管理也要从处理器的角度出发。由于处理器使用虚拟地址并通过映射来访问内存，这样内存管理就不能只关心内存的物理空间，还要关心虚拟地址空间，并对虚拟地址空间和映射关系进行管理。这样内核中实际的内存管理需求就包括了内存物理空间管理、虚拟地址空间管理，以及虚拟地址和内存物理地址映射关系的管理。

同样内存管理部分会有相应的性能需求，主要是减少碎片，包括外部碎片和内部碎片（与管理粒度相关）。另外对于分配和释放也有性能的要求（当然是越快越好）。

作为内核的基础功能之一，内存管理由于空间的限制，需要将各种模块及不同情况下对于内存的请求分为不同的优先级和不同的方式进行操作，这些也是内存管理需要满足的需求。

随着 SoC 芯片技术的发展，愈来愈多的协处理器加入到了 SoC 中。特别是视频应用需求的不断增长使得内存管理的需求也发生了变化，由于图像分辨率以及显示分辨率的不断提高，而协处理器通常又需要连续的物理内存，这样就有对于几兆甚至十几兆连续内存管理的

需求。而原有的 Linux 内核内存管理并不能满足这种需求，这样就需要开发新的内存管理方法。简便有效的方法是将相应的物理空间和 Linux 内核的空间保持分离，相应的物理空间单独进行管理，应用则通过映射来进行访问。这部分空间通常由芯片厂商提供管理方法，如 TI 的 CMEM、高通的 PMEM 等。当然随着内核的发展，芯片厂商各自为政的局面必然得到改观。现今内核也在逐渐整合这部分功能，提出了一些独立的方案，如 CMA 和 ION。最新的内核有对两者进行整合的趋势，很可能将 CMA 作为 ION 的基础。不管怎样，这个功能需求是固定的，原理也是基本一致的，后面会以 TI 的 CMEM 为例进行介绍，至于内核最终的解决形式还要时间的验证。

4.4.2　内存管理框架介绍

Linux 内核中的内存管理框架考虑到了各个方面的需求，实现得非常精细。在第 4.2 节中介绍了虚拟空间管理中内核地址映射的功能，只是内核空间的地址映射太重要而且太特殊了，所以单独作为一节来进行讨论。下面还会讨论虚拟空间管理，但是将会以用户空间为主。

1. 内核对于内存的管理和使用的整体框架

Linux 内核的内存管理也要满足内核自身的需要。管理同样会有粒度问题，作为现代操作系统通常地址映射都是以页为单位，这样进行物理内存管理以页为单位是比较合适的。内核各个模块所需要的数据大小不同，直接对页进行操作并不实际，并且容易产生碎片，所以需要更合理的分配方式。另外内核各个模块中又有很多固定大小的数据结构需要进行分配、释放。由于需要频繁的分配、释放相同类型的数据，所以一个好的管理方式为，将它们集中起来形成池，这样对于提高效率和减少碎片并且 cache 都是有好处的。这些都需要针对内核的内存管理提供完整的框架。图 4-32 展示了 Linux 内核的内存管理框架。

图 4-32 的最底层 page allocator 是对物理内存进行管理的模块，负责管理所有的物理内存，其分配和释放的都是以页（page）为单位、大小是 2^N 个连续的物理内存页。所有的内存管理都是以 page allocator 为基础的，其采用的算法为经典的 Buddy（伙伴算法），当然不排除以后会有更好的算法替代，目前来看 Buddy 还是很好地完成了任务。仅有好的页分配器是不能满足内核对于内存管理的需要的，前面已经介绍了，内核有很多频繁使用的数据结构，对于它们最好单独分配空间进行管理，这就形成了 SLAB 分配器（kmem_cache）。从接口名字 kmem_cache 可以看出该分配器还有作为数据结构 cache 的功能，其实将相同数据结构放到一起管理本身就减少了遍历内存区域的费时操作，从这个角度说相当于 cache 的功能。关于 SLAB 分配器，Linux 内核提供了 SLUB 和 SLOB 两种更轻便的分配器实现，相同的功能为不同类型的设备使用。解决了经常使用的数据结构，Linux 内核中还会有单独驱动使用的数据结构，以及那些并不经常使用的数据结构进行分配和管理的需求。为这些数据结构建立 kmem_cache 显然是一种浪费，但是 kmem_cache 又有这些优点，最好能够使用，这样内核针对应用就建立一组匿名的 SLAB cache，将一组特定大小的内存组织在一起形成 kmem_cache。由于并不是针对特定的数据结构而是特定大小（是 2^N 个字节，一般最小 16 B，最多 2 个 page，再大的话就使用页分配器了），所以是匿名的 SLAB cache，对这些空间分配的接口是 kmalloc。

以上讨论的内存管理接口获得的内存物理空间连续，并且进行映射时虚拟空间也是连续

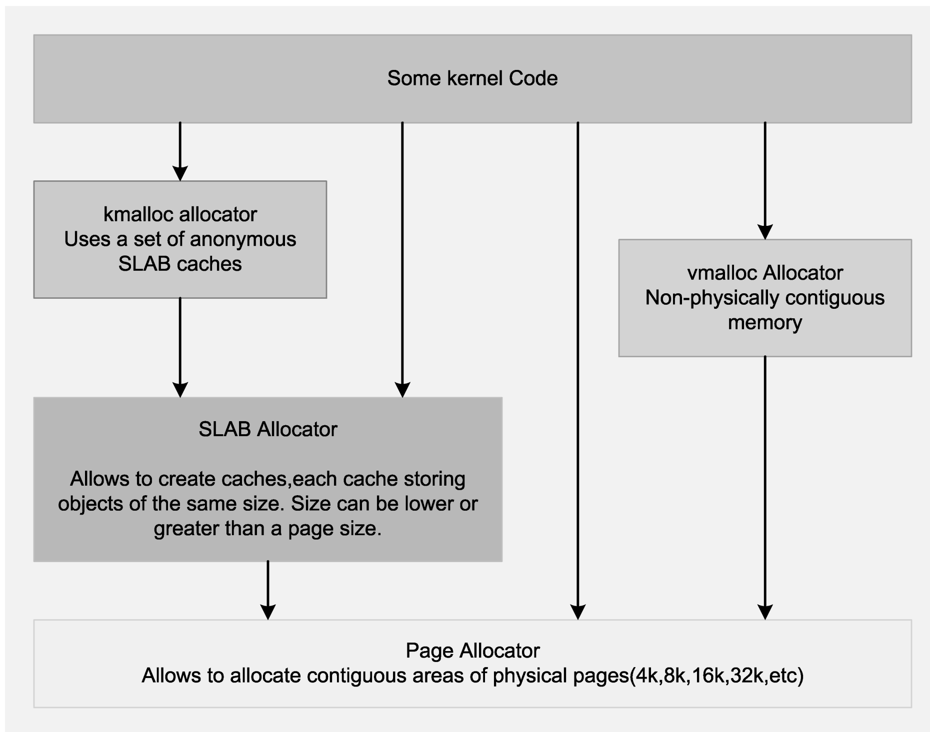


图 4-32　Linux 内核内存管理框架

的，具体到内核地址映射的区域是属于 low memory 的部分。内核部分只有这种内存管理模式是不够的，当 Linux 内核运行比较长的时间后仍要分配比较大的内存，如加载一个新的内核模块需要的空间等，此时很可能已经没有连续的物理内存了，而系统中还会有很多分散的内存（叫内存外部碎片）。这是任何算法也避免不了的，但仍要想办法将这部分内存空间利用起来。内核提供了相应的功能就是 vmalloc 分配器。vmalloc 分配器提供连续的虚拟地址，而对于物理页是可以不连续的，这样就在很大程度上解决了内存外部碎片的问题，也为各种内核模块提供了一种内存使用方法（如网络或文件系统中使用比较大的内存）。相应的内核虚拟空间就是 vmalloc 空间，注意其中使用的物理空间还是以页为单位的，自然相应的虚拟空间也会是页大小。

接下来看看 kmem_cache 和 vmalloc 的具体实现。首先看看 kmem_cache 的实现，对于 kmem_cache 的 slab 实现通常都是使用已经分配好的页面，但是需要的时候要对其使用的页面进行增长，就会调用 kmem_getpages 来获得新的物理页面直接供 slab 使用。看看 kmem_getpages 的具体实现：

```
static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
{
    struct page *page;
    int nr_pages;
    int i;
```

```
    …
        //根据 kmem_cache 的属性标记可回收属性
        flags |= cachep -> gfpflags;
        if(cachep -> flags & SLAB_RECLAIM_ACCOUNT)
            flags |= __GFP_RECLAIMABLE;

    //直接分配需要的所有连续物理页面,如果分配不到就直接失败
    page = alloc_pages_exact_node(nodeid, flags | __GFP_NOTRACK, cachep -> gfporder);
    if(!page)
            return NULL;

    //计算实际分配的页数
    nr_pages = (1 << cachep -> gfporder);
    //根据 kmem_cache 的属性标记统计物理页区域的可回收页
    if(cachep -> flags & SLAB_RECLAIM_ACCOUNT)
            add_zone_page_state(page_zone(page),
                NR_SLAB_RECLAIMABLE, nr_pages);
    else
            add_zone_page_state(page_zone(page),
                NR_SLAB_UNRECLAIMABLE, nr_pages);
    //设置所有相关的物理页管理实体,标记为 slab 使用
    for(i = 0; i < nr_pages; i++)
            __SetPageSlab(page + i);

        if(kmemcheck_enabled && !(cachep -> flags & SLAB_NOTRACK)) {
            kmemcheck_alloc_shadow(page, cachep -> gfporder, flags, nodeid);

            if(cachep -> ctor)
                kmemcheck_mark_uninitialized_pages(page, nr_pages);
            else
                kmemcheck_mark_unallocated_pages(page, nr_pages);
        }

        //直接返回 kmem_cache 需要的虚拟地址
        return page_address(page);
    }
```

从代码中可见，kmem_cache 在分配获得物理页后，并没有进行映射，而是直接就返回虚拟地址了。page_address 中只是根据物理页的管理实体返回正确的虚拟地址，并不进行地址映射的操作。那相应的地址映射是在哪里做的呢？答案是初始化的时候。由于其使用的是 low memory，而 low memory 在系统初始化的时候直接进行了地址映射，这样在使用相应空间时就不需要进行地址映射，从而大大地提高了系统效率。初始化时具体映射是通过 map_

lowmem 创建的。代码细节如下：

```
static void __init map_lowmem(void)
{
    struct memblock_region *reg;

    /* Map all the lowmem memory banks. */
    //对每个连续的内存块进行操作
    for_each_memblock(memory, reg) {
        phys_addr_t start = reg->base;
        phys_addr_t end = start + reg->size;
        struct map_desc map;

        if(end > lowmem_limit)
            end = lowmem_limit;
        if(start >= end)
            break;
        //这里有物理地址和虚拟地址,就是进行映射参数的设置
        map.pfn = __phys_to_pfn(start);
        map.virtual = __phys_to_virt(start);
        map.length = end - start;
        map.type = MT_MEMORY;
        //通过 create_mapping 映射 low memory 范围内的每块物理内存
        create_mapping(&map);
    }
}
```

接下来看看 vmalloc 的实现，看看有什么样的差别。vmalloc 是通过__vmalloc_node 建立的，从函数__vmalloc_node 的逻辑可了解 vmalloc 的机制。

```
static void *__vmalloc_node(unsigned long size, unsigned long align,
                gfp_t gfp_mask, pgprot_t prot,
                int node, void *caller)
{
    struct vm_struct *area;
    void *addr;
    unsigned long real_size = size;

    //大小要页对齐
    size = PAGE_ALIGN(size);
    if(!size || (size >> PAGE_SHIFT) > totalram_pages)
        return NULL;
```

```
    //这里先要从 vmalloc 的虚拟空间中,把虚拟地址范围(要满足大小)分好
    area = __get_vm_area_node(size, align, VM_ALLOC, VMALLOC_START,
                    VMALLOC_END, node, gfp_mask, caller);

    if(!area)
        return NULL;
    //根据虚拟空间的大小分配页面(每次分配一页获得需要数目的页面),并做映射
    addr = __vmalloc_area_node(area, gfp_mask, prot, node, caller);

    /*
     * A ref_count = 3 is needed because the vm_struct and vmap_area
     * structures allocated in the __get_vm_area_node() function contain
     * references to the virtual address of the vmalloc'ed block.
     */
    kmemleak_alloc(addr, real_size, 3, gfp_mask);

    return addr;
}
```

vmalloc 分配的物理页面究竟映射到哪里了？在__vmalloc_area_node 中会调用 map_vm_area 完成映射工作，而 map_vm_area 最终会调用 vmap_pud_range 来完成最终的映射工作，函数 vmap_pud_range 中可见相应的 vmalloc 映射的页目录来自哪里。

```
static int vmap_pud_range(pgd_t *pgd, unsigned long addr,
        unsigned long end, pgprot_t prot, struct page **pages, int *nr)
{
    pud_t *pud;
    unsigned long next;

    //这里从内核的虚拟空间和页根目录管理实体中分配页目录项
    pud = pud_alloc(&init_mm, pgd, addr);
    if(!pud)
        return -ENOMEM;
    //进行实际的物理页映射操作,会调用体系结构相关代码
    do {
        next = pud_addr_end(addr, end);
        if(vmap_pmd_range(pud, addr, next, prot, pages, nr))
            return -ENOMEM;
    } while(pud++, addr = next, addr != end);
    return 0;
}
```

从 vmap_pud_range 中可见，页根目录来自于 init_mm。在初始化时提到过 init_mm，但

是并没有进行详细介绍，现在可以详细说明了。对于页目录来说，由于用户空间是和任务相关的，所以用户空间每个任务都要有自己的页目录。而页目录本身从不同的体系结构来说，并不是都支持分离用户目录和内核目录的，为了保持一致性 Linux 内核使用页根目录来统一涵盖用户空间和内核空间。这就带来一个问题，对于所有的任务，进程的页目录中内核地址相关的部分应该是相同的，这该如何实现呢？最简单的方法就是在进程创建的时候进行。进程创建时会调用体系结构相关的 pgd_alloc，而 ARM 体系结构下会将其定义为 get_pgd_slow，这里就分配页根目录，并将内核的地址空间映射进行复制。

```
pgd_t *get_pgd_slow(struct mm_struct *mm)
{
    pgd_t *new_pgd, *init_pgd;
    pmd_t *new_pmd, *init_pmd;
    pte_t *new_pte, *init_pte;
    //ARM 体系结构要求页根目录(一级页表)大小为 16K,所以 order 为 2
    new_pgd = (pgd_t *)__get_free_pages(GFP_KERNEL, 2);
    if(!new_pgd)
        goto no_pgd;
    //这里是保护,需要将内核地址空间重新初始化
    memset(new_pgd, 0, FIRST_KERNEL_PGD_NR * sizeof(pgd_t));

    /*
     * Copy over the kernel and IO PGD entries
     */
    //获得内核地址空间的页根目录
    init_pgd = pgd_offset_k(0);
    //复制内核地址空间的页目录项内容到新任务进程的对应地址页目录项中
    memcpy(new_pgd + FIRST_KERNEL_PGD_NR, init_pgd + FIRST_KERNEL_PGD_NR,
            (PTRS_PER_PGD - FIRST_KERNEL_PGD_NR) * sizeof(pgd_t));

    clean_dcache_area(new_pgd, PTRS_PER_PGD * sizeof(pgd_t));

    //对于向量表是低地址的情况,由于起始地址是 0x0,所以需要进行特别的映射操作
    if(!vectors_high()){
        /*
         * On ARM, first page must always be allocated since it
         * contains the machine vectors.
         */
        new_pmd = pmd_alloc(mm, new_pgd, 0);
        if(!new_pmd)
            goto no_pmd;

        new_pte = pte_alloc_map(mm, new_pmd, 0);
```

```
            if(!new_pte)
                goto no_pte;

            init_pmd = pmd_offset(init_pgd, 0);
            init_pte = pte_offset_map(init_pmd, 0);
            set_pte_ext(new_pte, *init_pte, 0);
            pte_unmap(init_pte);
            pte_unmap(new_pte);
        }

        return new_pgd;

no_pte:
        pmd_free(mm, new_pmd);
no_pmd:
        free_pages((unsigned long)new_pgd, 2);
no_pgd:
        return NULL;
}
```

对应内核的页根目录（一级页表）的获得是通过宏 pgd_offset_k 来完成的，来看看相应的定义：

```
/* to find an entry in a kernel page-table-directory */
#define pgd_offset_k(addr)pgd_offset(&init_mm, addr)
```

又见到 init_mm，从这个宏中可以明白，init_mm 就是管理整个内核虚拟地址空间的实际管理实体。相应的也会包含最基本的内核地址空间映射的页根目录。看看 init_mm 的定义：

```
struct mm_struct init_mm = {
        .mm_rb          = RB_ROOT,
        .pgd            = swapper_pg_dir,
        .mm_users       = ATOMIC_INIT(2),
        .mm_count       = ATOMIC_INIT(1),
        .mmap_sem       = __RWSEM_INITIALIZER(init_mm.mmap_sem),
        .page_table_lock =  __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
        .mmlist         = LIST_HEAD_INIT(init_mm.mmlist),
        .cpu_vm_mask    = CPU_MASK_ALL,
        INIT_MM_CONTEXT(init_mm)
};
```

其中的 pgd 就是页根目录，相应的值就是 swapper_pg_dir。对 swapper_pg_dir 应该并不陌生，在讲地址映射中已经提到了，就是页根目录（一级页表）的地址。ARM 体系结构是

在物理内存的 16 KB ~ 32 KB 空间。

关于 vmalloc 空间还是要多说一些，在 vmalloc 空间中相应的映射并不是像 low memory 那样只要初始化的时候做一次即可，而是需要在运行时动态增减。而内核当前使用的页目录可能是任何任务进程的页目录，这就需要 vmalloc 动态创建的映射最终也能保证一致性。很直接的方法是在创建 vmalloc 映射时，更新所有进程相应的内核地址空间页目录项，但是很低效，而内核只有在需要时（通过缺页异常实现）才对进程的内核地址空间相关目录项进行同步操作。具体实现中可见，vmalloc 中实际的映射操作还是针对 init_mm 进行，保证内核地址空间管理实体的正确性。当内核使用那些没有同步的进程页表访问相应的 vmalloc 空间时会发生 data abort，在 ARM 体系结构下，相应的 data abort 最终会调用 do_translation_fault。在 do_translation_fault 中对于内核地址 vmalloc 空间发生的 data abort 最终会执行下面的语句：

```
//根据地址获得页目录项偏移
index = pgd_index(addr);

/*
 * FIXME: CP15 C1 is write only on ARMv3 architectures.
 */
//根据处理器的寄存器获得要操作的页目录项
pgd = cpu_get_pgd() + index;
//获得内核地址空间的管理实体的页目录项
pgd_k = init_mm.pgd + index;

if(pgd_none(*pgd_k))
    goto bad_area;

if(!pgd_present(*pgd))
    set_pgd(pgd, *pgd_k);
//获得二级页表
pmd_k = pmd_offset(pgd_k, addr);
pmd   = pmd_offset(pgd, addr);

index = (addr >> SECTION_SHIFT) & 1;
if(pmd_none(pmd_k[index]))
    goto bad_area;

//实际将二级页表进行复制
copy_pmd(pmd, pmd_k);
return 0;
```

至此 Linux 内核的虚拟地址空间就保证了一致性。当然由于 Linux 内核在切换进程时要切换一级页表，所以相同 vmalloc 的访问，可能发生不止一次的访问异常。但是这些都是在需要的时候才进行，相对来说开销较小，是一种高效的实现方式。

以上介绍了内核中主要的使用内存的方法及相关的细节。下面总结一下，对内存的使用，基本的步骤就是要通过 Page Allocator 获得需要的页面，并需要有内核地址空间的映射。复杂的映射流程会影响系统效率，所以对于 kmem_cache 使用的是 low memory 部分的线性映射，采用的办法是在初始化时映射，减少系统运行时的映射步骤，从而提高效率；而 vmalloc 使用的是 vmalloc 空间的映射方式，通过对分散物理页面运行时的映射减少外部碎片；另外还有大容量内存的 high mem 部分的 kmap 映射方式也是可行的。用户空间内存的使用，内核采用类似于 vmalloc 的方式进行，首先获得虚拟地址空间，后分配页，然后再映射到用户的地址空间，只是用户空间需要的物理内存页，并不能直接满足用户的最大需求，而是采取最小分配和尽量延迟实际分配的方法，只有在不得不进行实际物理空间的分配时才会进行分配。这其中不仅涉及物理内存管理还涉及地址映射相关的处理器异常，这些紧密结合才能完成相应的工作。用户虚拟空间的管理及内存使用，本节的后面会有详细介绍。

2. 内核对于物理内存管理的整体框架

从前面的介绍可以看出，物理内存的管理主要的工作都集中在 Page Allocator 上。Page Allocator 负责内存的组织和管理。说到内存的组织和管理，就既要考虑用户空间的需求同样要考虑内核空间的需求，对内核来说效率是很重要的一部分功能。如何在内存管理上提高效率呢？这就要回过头来考虑映射，对于内核来说进行线性映射是提高效率的重要方法，这样对于频繁访问的数据，无论需要物理地址还是虚拟地址都可以进行直接的线性运算来完成，而不需要进行页表的查询以及负责的结构进行转换，所以对内核使用高频的数据进行线性映射是提高效率的好方法。而内核地址空间是有限的，且现有的内存基本都超出了 32 位的内核地址空间的大小，所以需要对物理内存进行合理的组织来满足内核以及特殊设备的需求。图 4-33 展示了 Linux 内核如何组织和管理物理内存的。

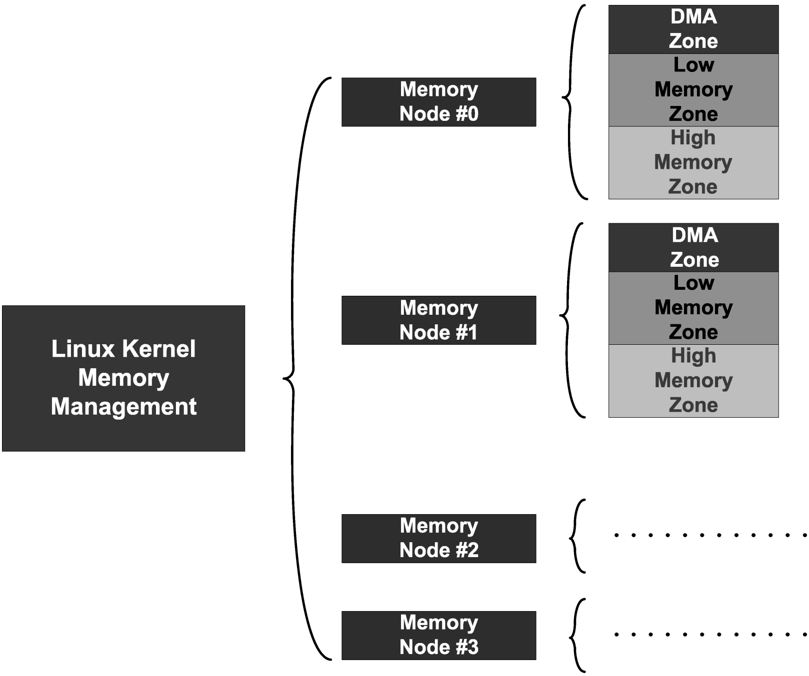


图 4-33　Linux 内核物理内存的组织框架

从图 4-33 中可见，Linux 内核是如何对物理内存进行组织的。Linux 内核的这种内存组织适用于各种体系结构。首先看到的是 memory node，每个 memory node 就是一个或者一组 cpu（SMP）可以访问的本地内存。如果多个 memory node 就表示有多组 cpu 每组都有本地内存，而多组 cpu 以及相应的本地内存通过高速总线互连，对应的体系结构属于 NUMA 架构，对应于某个 cpu 来说访问不同的 memory node 的成本是不同的，所以 memory node 其中有访问成本的概念在里面。而对于普通的 SoC 来说只有一个 memory node（这属于 UMA 架构），也就是说无论是单 cpu 还是 SMP，只有本地内存。对于本地内存则要根据需求划分不同的区域即 zone。zone 的划分是和需求分不开的，主要还是为了相应的效率和内核映射以及设备的各种需求。首先是 DMA zone 这部分区域，有该区域的原因主要是由于某些处理器的 DMA 访问空间有物理地址的限制，需要将这部分限制的物理空间单独形成一个区域就是 DMA zone。通常只有老的设备有该限制，ARM 体系结构中新的内核已经没有该区域，但是 Linux 内核为了广泛的适用范围还是保留该区域。接下来是 low memory zone 即 ZONE_NORMAL，这部分的物理内存空间就是可以进行线性映射的空间。在 low memory zone 之后是 high memory zone，对于 high memory zone 中的物理内存页可以映射到 vmalloc 以及 pkmap 的地址空间。对于使用 DMA zone 和 high memory zone 分别需要配置 CONFIG_ZONE_DMA 和 CONFIG_HIGHMEM，ARM 体系结构中，DMA 并没有限制，所以不需要配置 CONFIG_ZONE_DMA，而 CONFIG_HIGHMEM 的配置和物理内存的大小相关，通常超过 768 MB 内存就可以配置 CONFIG_HIGHMEM。

内核如何区分 low memory 和 high memory 的大小呢？又要回到初始化的阶段，在函数 sanity_check_meminfo 中可见以下代码：

```
#ifdef CONFIG_HIGHMEM
        if(__va(bank->start) > vmalloc_min ||
            __va(bank->start) < (void *)PAGE_OFFSET)
            highmem = 1;

        bank->highmem = highmem;

        /*
         * Split those memory banks which are partially overlapping
         * the vmalloc area greatly simplifying things later.
         */
        if(__va(bank->start) < vmalloc_min &&
            bank->size > vmalloc_min - __va(bank->start)) {
            if(meminfo.nr_banks >= NR_BANKS) {
                printk(KERN_CRIT "NR_BANKS too low, "
                        "ignoring high memory\n");
            } else {
                memmove(bank + 1, bank,
                    (meminfo.nr_banks - i) * sizeof(*bank));
```

```
                meminfo. nr_banks ++;
                i ++;
                bank[1]. size  - = vmalloc_min - __ va(bank -> start);
                bank[1]. start = __ pa(vmalloc_min - 1) + 1;
                bank[1]. highmem = highmem = 1;
                j ++;
            }
            bank -> size = vmalloc_min - __ va(bank -> start);
        }
#else
        bank -> highmem = highmem;

        /*
         * Check whether this memory bank would entirely overlap
         * the vmalloc area.
         */
        if(__ va(bank -> start) >= vmalloc_min ||
            __ va(bank -> start) < (void *)PAGE_OFFSET){
            printk(KERN_NOTICE "Ignoring RAM at %. 8lx - %. 8lx "
                    "(vmalloc region overlap). \n",
                    bank -> start, bank -> start + bank -> size - 1);
            continue;
        }

        /*
         * Check whether this memory bank would partially overlap
         * the vmalloc area.
         */
        if(__ va(bank -> start + bank -> size) > vmalloc_min ||
            __ va(bank -> start + bank -> size) < __ va(bank -> start)){
            unsigned long newsize = vmalloc_min - __ va(bank -> start);
            printk(KERN_NOTICE "Truncating RAM at %. 8lx - %. 8lx "
                    "to  - %. 8lx(vmalloc region overlap). \n",
                    bank -> start, bank -> start + bank -> size - 1,
                    bank -> start + newsize - 1);
            bank -> size = newsize;
        }
#endif
```

从这段代码可见，只有在设置 CONFIG_HIGHMEM 时才会有 high memory 的区域，主要的功能是使得 low memory 的大小在进行直接线性映射后不能进入 vmalloc 的空间，这样保证映射的正确性，如果定义了 CONFIG_HIGHMEM，则将超出的部分归入 high memory 的区域，

否则忽略相应的物理空间。从代码可见 vmalloc 的设置对于 low memory 所占的空间是有影响的，通常只有物理内存的容量大并且 CONFIG_HIGHMEM 设置才会有 high memory 的空间。当然可以在 CONFIG_HIGHMEM 设置的情况下，通过启动参数“vmalloc =”改变 vmalloc 映射空间的大小，进而来改变 low memory 所占空间的大小。当然也不能任意减少 low memory 的空间，至少还要给 low memory 保留 32 MB 的空间。对 vmalloc 空间的限制可在 early_vmalloc 中找到。

Linux 系统中 normal zone 是必需的，而 high zone 的使用与否是和物理内存的大小相关的。拥有大物理内存的系统通过配置 CONFIG_HIGHMEM 可以使能相应的功能，并使用相应的区域。嵌入式设备，特别是拥有视频能力的设备通常都有比较大的内存，就需要配置 CONFIG_HIGHMEM 来使用 high zone 的空间。其实，内核留了一个门，可以在内存并不多的情况下使用 high zone 的方式进行内存管理。这个门就是前面通过配置 CONFIG_HIGHMEM 并对 vmalloc 进行设置来实现的，当 vmalloc 的设置 size 足够大，而压缩的 low memory 空间只要小于物理内存的大小，就会有一部分空间放入 high zone 的管理区域。但是要明确两个空间的管理效率是不同的，low memory 的减少会造成映射开销的加大，从而降低效率，特别是内核需要很多空间来实现数据结构的 cache 功能。所以在内存不足够大的情况下没有必要规划 high zone 的空间。

normal zone 和 high zone 的物理空间究竟是如何使用的，如图 4-34 所示。

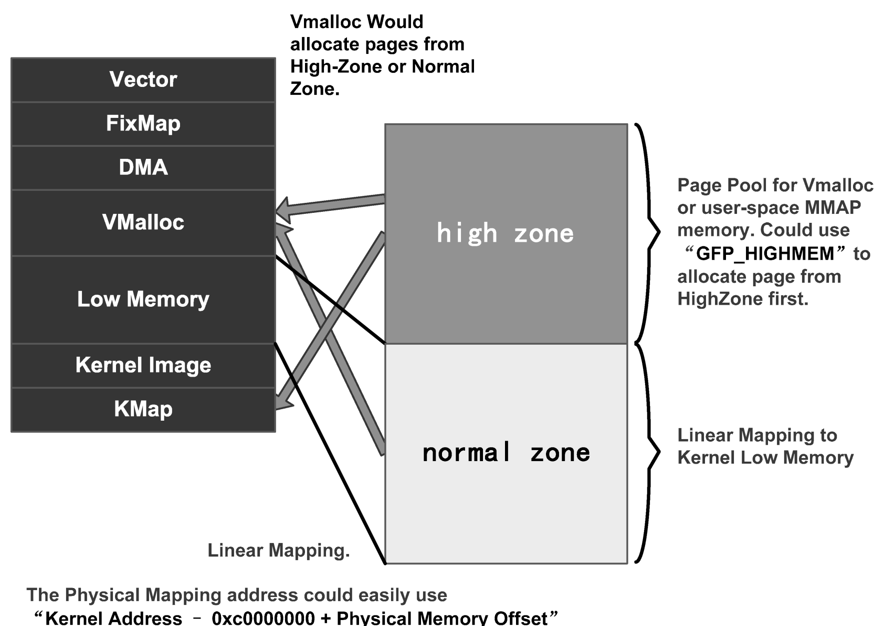


图 4-34　Linux 内核物理内存 zone 的使用框图

图 4-34 左侧的部分是以 ARM 体系结构为基础的内核映射空间分配框图，从中可见 normal zone 的空间主要还是为 low memory 这种线性映射服务的，但是同样还会为 vmalloc 服务。而 high zone 则不会映射到 low memory 的空间，对内核来说是作为补充使用。vmalloc 的空间中分配物理内存的操作既可以从 high zone 的空间也可以从 normal zone 的空间获取，如果两个空间都存在的话，分配就要有一个先后顺序，这个顺序就是：先使用 high zone 的空间，

然后再使用 normal zone 的空间，原因在于 normal zone 对内核本身来说是效率最高的物理空间，相关的空间在可以的情况下尽量留给内核使用。注意用户空间需要的物理内存分配同样会进行所需区域的标记，只是这种标记是通过属性宏设定的。

对物理内存的分配，如何标记这种区域的顺序呢？比如说先从 high zone 的内存区域进行分配等，对物理内存管理区域的选择通过 gfp_zone 和 first_zones_zonelist 来实现，详细的代码如下：

```
static inline enum zone_type gfp_zone(gfp_t flags)
{
    enum zone_type z;
    int bit = (__force int)(flags & GFP_ZONEMASK);

    z = (GFP_ZONE_TABLE >> (bit * ZONES_SHIFT)) &
                ((1 << ZONES_SHIFT) - 1);

    if(__builtin_constant_p(bit))
        MAYBE_BUILD_BUG_ON((GFP_ZONE_BAD >> bit) & 1);
    else {
#ifdef CONFIG_DEBUG_VM
        BUG_ON((GFP_ZONE_BAD >> bit) & 1);
#endif
    }
    return z;
}

static inline struct zoneref *first_zones_zonelist(struct zonelist *zonelist,
                    enum zone_type highest_zoneidx,
                    nodemask_t *nodes,
                    struct zone **zone)
{
    return next_zones_zonelist(zonelist->_zonerefs, highest_zoneidx, nodes, zone);
}
```

其中 first_zones_zonelist 会获得一个物理内存区域的数组，按照相应的顺序来试图获得物理页，而 gfp_zone 是检查物理内存需求属性，根据相应的属性获得首先应该检查的物理内存区域。这里涉及一个重要的宏定义就是 GFP_ZONE_TABLE，其内容如下：

```
#define GFP_ZONE_TABLE( \
(ZONE_NORMAL << 0 * ZONES_SHIFT)          \
 | (OPT_ZONE_DMA << ___GFP_DMA * ZONES_SHIFT)          \
 | (OPT_ZONE_HIGHMEM << ___GFP_HIGHMEM * ZONES_SHIFT)          \
```

```
|(OPT_ZONE_DMA32 << __GFP_DMA32 * ZONES_SHIFT)          \
|(ZONE_NORMAL << __GFP_MOVABLE * ZONES_SHIFT)           \
|(OPT_ZONE_DMA << (__GFP_MOVABLE | __GFP_DMA) * ZONES_SHIFT) \
|(ZONE_MOVABLE << (__GFP_MOVABLE | __GFP_HIGHMEM) * ZONES_SHIFT) \
|(OPT_ZONE_DMA32 << (__GFP_MOVABLE | __GFP_DMA32) * ZONES_SHIFT) \
)
```

从中可以看到很多以__GFP 开头的属性说明，这些属性都是在内存管理中，对应着不同的内存管理区域，如__GFP_DMA、__GFP_HIGHMEM 和__GFP_DMA32。对于 GFP_ZONE_TABLE，所在位越低，就越重要，所以 ZONE_NORMAL 是在最低位。对于以__GFP 开头的属性中，这里没有看到__GFP_NORMAL，因为对于 Linux 内核来说不标记就是使用 normal zone 的区域。另外需要注意的是 OPT_ZONE_DMA、OPT_ZONE_HIGHMEM 和 OPT_ZONE_DMA32 这些宏，由于取出 OPT 相应的区域是否存在都是和内存配置相关的，而 GFP_ZONE_TABLE 本身应该不受内核配置的影响，这就需要通过这些宏来解决。比如说没有任何配置的情况下 OPT_ZONE_DMA 和 OPT_ZONE_HIGHMEM 实际都是配置成 ZONE_NORMAL。最后可见特殊的属性宏__GFP_MOVABLE，为什么会有该宏呢？这是由于很长时间以来，物理内存的碎片一直是 Linux 的弱点之一。尽管提出了很多方法，但一直没有合适的方法能够既满足 Linux 对各种类型工作的性能需求，同时又对其他模块影响最小。直到内核 2.6.24 开发期间，防止物理内存碎片的方法终于加入内核。相应的方法受到文件系统碎片处理的启发，文件系统也有碎片，其碎片问题主要通过碎片合并工具解决，分析文件系统，重新排序已分配的存储块，从而建立较大的连续存储区。理论上，该方法对物理内存也是可行的，但相应的方法需要物理页是可移动的，这样才能通过移动来解决碎片问题。但是由于内核使用的物理内存页通常是不能移动的，所以要通过该方法解决碎片问题，就要将物理页分开考虑，分为可移动的和不可移动的。总的来说，内核减少物理内存碎片的方法是试图从开始就尽可能防止碎片。

内核的反碎片方法，首先是要按照物理页的属性分为不同的类型。对 Linux 内核来说，主要是三种不同类型：

① 不可移动页。在内存中有固定位置，不能移动到其他地方。内核核心分配的大多数内存属于该类别。

② 可移动页。可以随意地移动。用户空间应用程序的页属于该类别。这类页是通过动态页表映射的。如果它们复制到新位置，只要相应的更新页表项即可，而应用程序是不会注意到发生的事情。

③ 可回收页。也属于可移动页，只是其不能直接移动，但是其内容由于可以从某些源来重新生成，所以其页面可以直接释放，来释放物理内存页，然后重新分配物理页再恢复，这样就是逻辑意义上的可移动。映射的数据页（其映射源是文件系统的文件）就属于该类别。如果有 swap 分区，内核的 kswapd 守护进程会根据可回收页访问的频繁程度，周期性释放此类内存。这是一个复杂的过程，只要了解内核会在需要的时候进行可回收页的回收来释放内存就可以了。

反碎片技术就是在分配的时候考虑到这些页的属性，比如不可移动的页不能位于可移动

内存区的中间，否则就无法从该内存区域获得较大的连续物理内存；而可移动的内存页，由于最终可以通过移动页面来减少碎片，则相应的限制就比较少。基本的思路如此，具体的算法细节就不讨论了。__GFP_MOVABLE 就是相应的可移动属性的标识。需要注意的是，这些属性都是附加在物理区域上的逻辑属性，这些逻辑属性的页面本身也可以组织成附加的区域，相应的 Linux 内核增加了 ZONE_MOVABLE 来表示其中的页面都是可移动的。Linux 内核中可移动区域可以通过启动参数 kernelcore 或者 movablecore 进行设置，两者的差别是前者设置的是不可移动区域空间的大小，而后者设置的是可移动区域空间的大小。

前面看到了和物理内存区域相关的内存分配属性宏。物理内存管理是 Linux 内核的基础，是各种模块需要的底层模块之一，所以相应的分配属性宏不只和区域相关，还和各种模块以及分配优先级相关的属性，相应的属性说明如下：

- __GFP_WAIT 表示分配内存的请求可以中断。也就是说，调度器在该请求期间可选择另一个进程执行，或者该请求可以被另一个更重要的事件中断。分配器还可以在返回内存之前，在队列上等待一个事件（相关进程会进入睡眠状态）。
- __GFP_HIGH 表示请求非常重要，内核急切地需要内存时设置该标识，在分配内存失败可能给内核带来严重后果时（比如威胁到系统稳定性或系统崩溃），就会使用该标志。注意这个 high 并不是 high zone，虽然名字相似，但__GFP_HIGH 与__GFP_HIGH-MEM 毫无关系，请不要弄混这两者。
- __GFP_IO 说明在查找空闲内存期间内核可以进行 I/O 操作。实际上，这意味着如果内核在内存分配期间需要换出页，那么只有当设置该标识时，才能将选择的页写入硬盘。
- __GFP_FS 允许内核执行 VFS 操作。在与 VFS 层有联系的内核子系统中必须禁用该标识，因为这可能引起循环递归调用导致死锁。
- __GFP_NOWARN 在分配失败时禁止内核故障警告，在极少数场合该标志有用。
- __GFP_REPEAT 在分配失败后自动重试，但在尝试若干次之后会停止。
- __GFP_NOFAIL 在分配失败后一直重试，直至成功。
- __GFP_ZERO 在分配成功时，将返回页填充字节 0。
- __GFP_HARDWALL 只在 NUMA 系统上有意义。它限制只在分配到当前进程的各个 CPU 所关联的结点分配内存。如果进程允许在所有 CPU 上运行（默认情况），该标志是无意义的。只有进程可以运行的 CPU 受限时，该标志才有效果。
- __GFP_THISNODE 也只在 NUMA 系统上有意义。如果设置该比特位，则内存分配失败的情况下不允许使用其他结点作为备用，需要保证在当前结点或者明确指定的结点上成功分配内存。
- __GFP_RECLAIMABLE 和__GFP_MOVABLE 是前面说明的减少碎片机制的属性标识。顾名思义，它们分别将分配的内存标记为可回收的或可移动的。

这些是属于底层的属性标识，而对于内核各个模块通常使用的属性则是以上属性的组合，具体的信息见 gfp. h，常用的重要的属性宏定义如下：

```
#define GFP_ATOMIC (__GFP_HIGH)
#define GFP_NOIO    (__GFP_WAIT)
```

```
#define GFP_NOFS (__GFP_WAIT | __GFP_IO)
#define GFP_KERNEL (__GFP_WAIT | __GFP_IO | __GFP_FS)
#define GFP_TEMPORARY(__GFP_WAIT | __GFP_IO | __GFP_FS | \ __GFP_RECLAIMABLE)
#define GFP_USER   (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
#define GFP_HIGHUSER   (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL | \ __
            GFP_HIGHMEM)
#define GFP_HIGHUSER_MOVABLE   (__GFP_WAIT | __GFP_IO | __GFP_FS | \__GFP_HARD-
            WALL | __GFP_HIGHMEM | \__GFP_MOVABLE)
#define GFP_IOFS(__GFP_IO | __GFP_FS)
```

不同区域的物理内存管理是通过伙伴算法实现的，具体如图 4-35 所示。

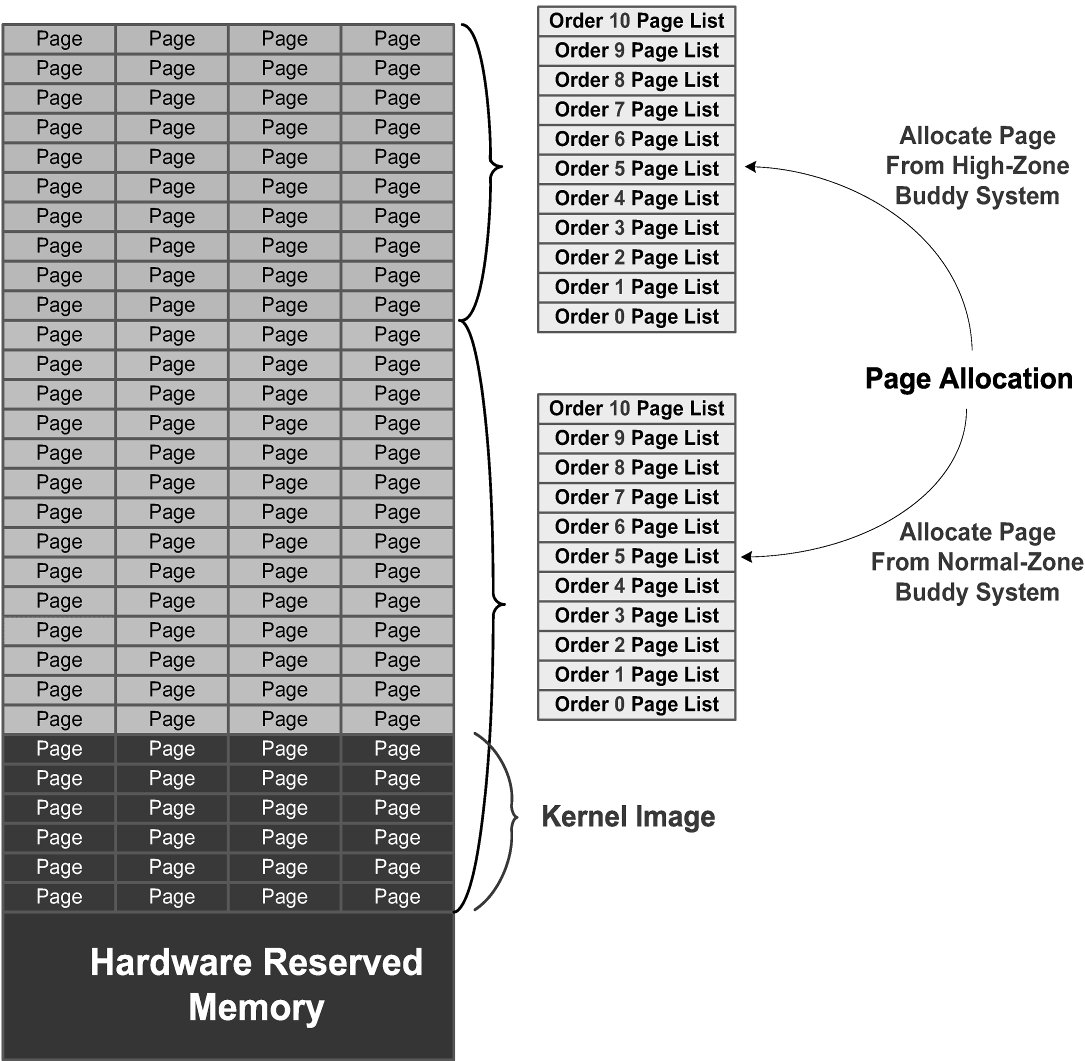


图 4-35　Linux 内核物理内存伙伴算法系统框图

由图 4-35 可见，整个 Linux 内核的内存管理框架是以区域进行管理的，每个区域内部使用伙伴系统（buddy system）将物理页以 2 的 N 次幂的连续物理页进行管理，每个幂次单独组成空闲链表。对某个内存区域的物理页分配就是找到合适的幂次链表，分配相应的连续物理页，如果相应幂次空闲链表是空，则从高幂次的空闲链表中拆分出合适的物理页进行分配。当然为了实现反碎片技术，伙伴系统也是要进行相应的修改，主要是在每个区域中也加入页类型的管理，相应的空闲链表也按照相应的属性组织，如图 4-36 所示。从图 4-36 可见，每个内存区域中不同幂次的空闲链表实际是数组，每个项是一种类型的页面，这样在进行分配和释放的时候都可以考虑避免内存碎片的问题，从而实现反碎片的完整方案。具体的算法就不详细讨论了。

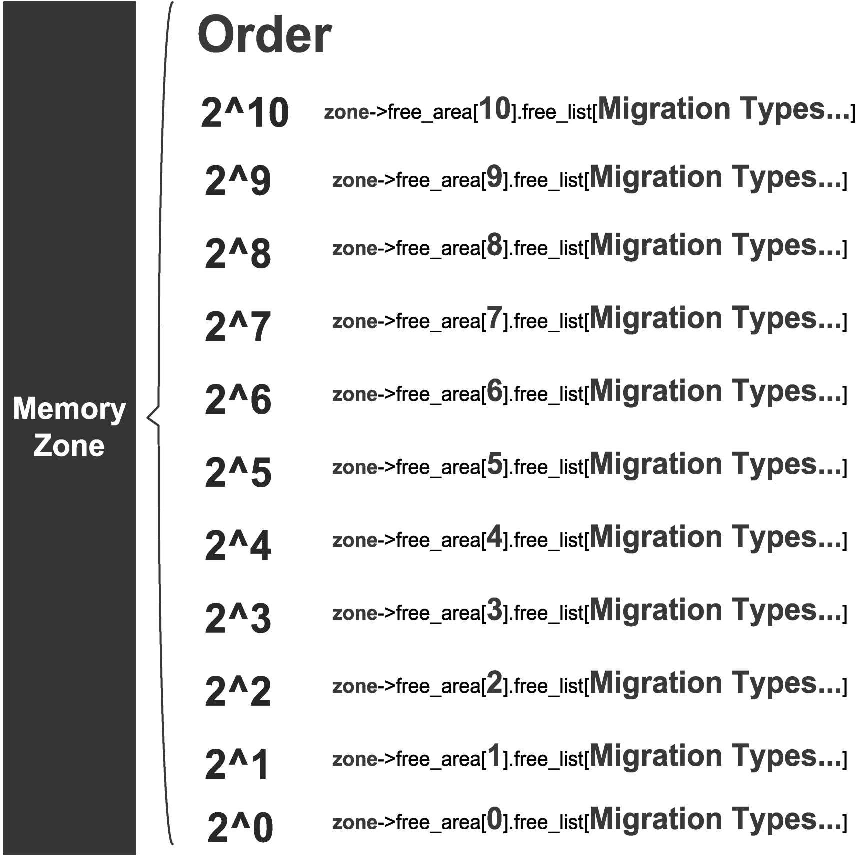


图 4-36　内存 zone 管理反碎片实现

物理内存的管理还要考虑的部分就是初始化的流程，图 4-37 中可见详细的物理内存管理初始化流程。

从图 4-37 中可见，主要的物理内存管理初始化是分为两部分进行操作的，第一部分主要是建立区域的信息即内存 node 及其中的 zone 的信息，而具体的空闲页管理并没有进行相应的初始化，在这一部分中主要是根据启动参数以及内核体系结构相关的配置，将总的内存信息进行整理形成合适的区域，另外系统会拿出一部分页来供初始化分配器 bootmem 使用，该分配器很简单，使用位图管理，进行初始化阶段内存管理，满足初始化第二部分操作完之

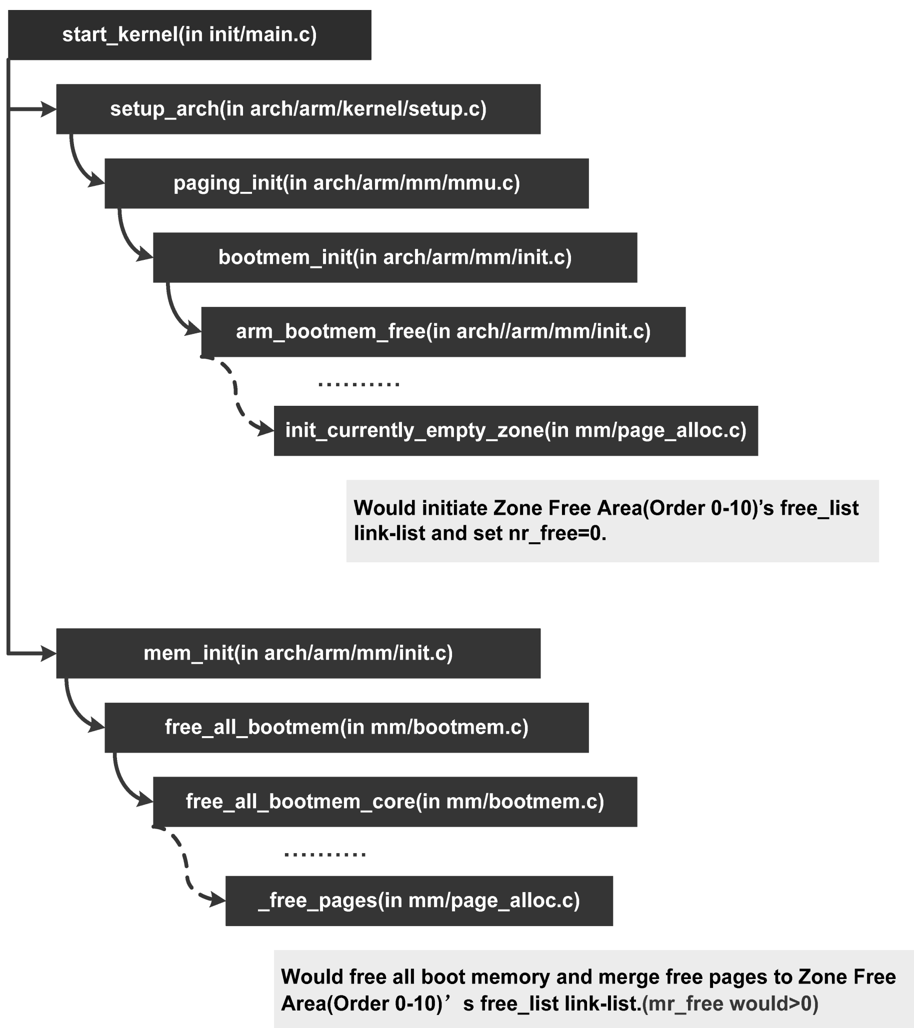


图 4-37　Linux 内核内存管理初始化流程图

前内核特别的分配内存的需要（如启动命令行的保存等）；第二部分是建立每个内存区域的空闲页管理信息，根据之前关于初始化的介绍，这时已经有完整的系统内存信息，并建立了相应的区域列表，用于实际的内存分配系统，此部分的主要工作就是释放相应的 bootmem 初始化分配器分配的空间，并根据物理内存的使用情况建立每个区域的空闲页管理链表。在第二部分完成之后，整个系统就可以使用物理内存管理系统（buddy system）进行内存管理了。

为什么要分两部分进行初始化呢？主要是由于 Linux 内核不是只支持简单的单 CPU 单内存的框架，还要支持多 CPU 和 NUMA 等复杂的框架，这样会对物理内存有不同的需求。面对如此复杂的系统，将物理内存管理分开两部分进行，就可以适应更复杂的情况。如多 CPU 需要每个 CPU 分配一定的物理页作为 CPU 特有内存以提高效率；NUMA 要了解不同的内存节点及其区域以了解完整的系统物理内存管理布局。另外分成两部分也方便未来系统功能增强和扩展。

3. 内核对于虚拟地址管理的整体框架

了解了物理内存空间管理之后，就要进入虚拟地址空间了。相应的 Linux 内核提供了完整的虚拟地址管理框架。关于虚拟地址空间，在地址映射时主要介绍了内核空间，32 位系统完整映射如图 4-38 所示。

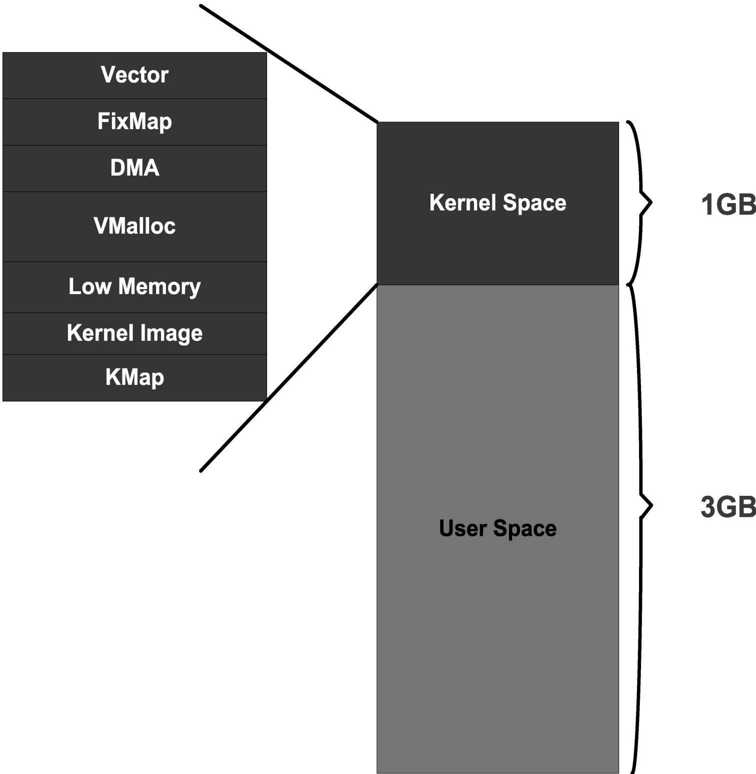


图 4-38　32 位系统完整映射

从图 4-38 可见，32 位系统内核通常的分配是 3 - 1 分配（也可通过配置修改成 2 - 2 分配）即 3 GB 的用户空间，1 GB 的内核空间。关于内核虚拟地址空间映射部分在第 4. 2 节已经进行了详细的介绍，接下来主要介绍用户空间的部分是如何管理的。

关于用户虚拟地址空间，首先明确它是运行用户的应用程序形成的，所以与用户程序是息息相关的。考虑一下用户空间都包含什么功能的数据？其中要加载可执行文件，而可执行文件就有不同的区域划分，如代码段、数据段、bss 段等，这些都应该在用户空间中有所体现。另外对于用户空间内存管理堆和栈都是必不可少的，也要有所体现。再有一部分就是文件映射等相关的映射部分。针对用户虚拟地址空间，内核的整体分布如图 4-39 所示。

从图 4-39 可见，代表用户程序的进程管理实体 task_struct 结构中，有对于虚拟地址空间管理的结构实体 mm_struct（内核虚拟地址空间也有相应结构的管理实体 init_mm），整个的用户虚拟地址空间都在 mm_struct 的掌握之中。不同的进程必然由不同的 mm_struct 来管理它的虚拟地址空间。下面来看看结构 mm_struct 的具体内容：

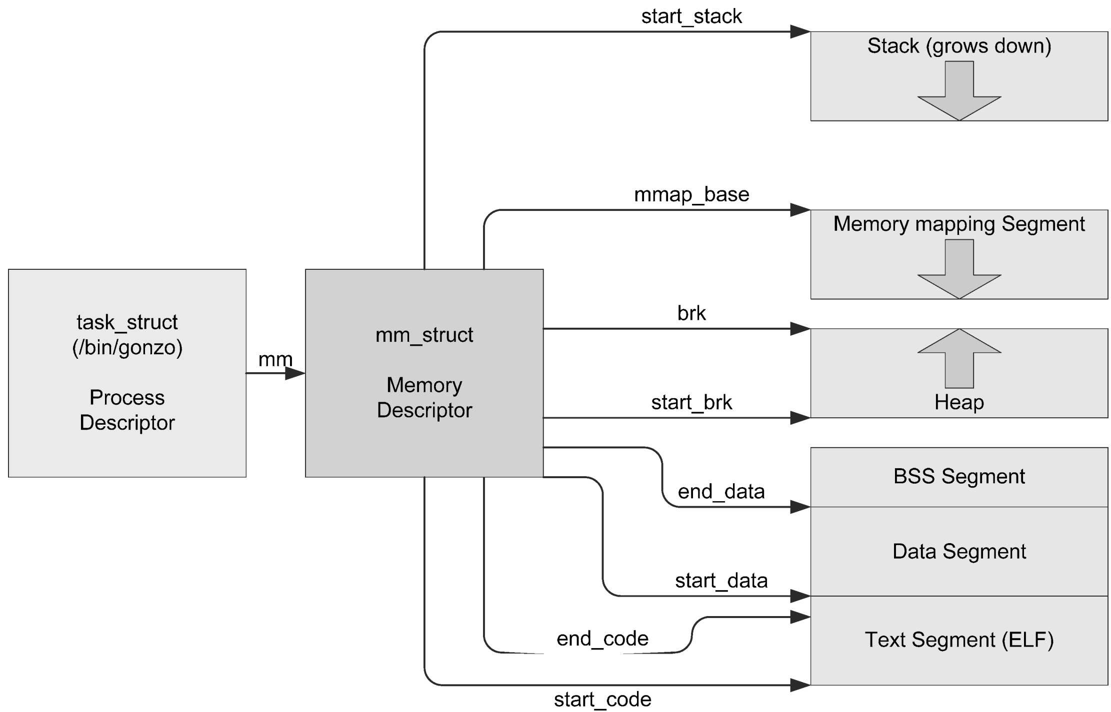


图 4-39　用户虚拟地址空间整体管理分布

```
struct mm_struct {
struct vm_area_struct * mmap;          /* list of VMAs */
struct rb_root mm_rb;
struct vm_area_struct * mmap_cache; /* last find_vma result */
...
unsigned long mmap_base;               /* base of mmap area */
unsigned long task_size;               /* size of task vm space */
unsigned long cached_hole_size;   /* if non-zero, the largest hole below free_area_cache */
unsigned long free_area_cache;         /* first hole of size cached_hole_size or larger */
pgd_t * pgd;
atomic_t mm_users;                     /* How many users with user space? */
atomic_t mm_count;        /* How many references to "struct mm_struct" (users count as 1) */
int map_count;                         /* number of VMAs */
struct rw_semaphore mmap_sem;
spinlock_t page_table_lock;            /* Protects page tables and some counters */
...
unsigned long hiwater_rss;             /* High-watermark of RSS usage */
unsigned long hiwater_vm;              /* High-water virtual memory usage */

unsigned long total_vm, locked_vm, shared_vm, exec_vm;
unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
```

```
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;

    ...
        struct linux_binfmt *binfmt;
        cpumask_t cpu_vm_mask;
        /* Architecture - specific MM context */
        mm_context_t context;
    ...
    };
```

从 mm_struct 的结构中可见，其中包含了进程虚拟地址空间的管理属性（如锁、引用计数等），以及各个部分的地址快速索引。但是只有地址是不够的，进程的虚拟地址空间的每个部分的数据属性及其操作方法也是不同的，同样需要相应的管理实体来表示。另外仅有虚拟地址的管理也是不够的，进程相应的空间还是要映射到实际的物理内存中。这就是 mm_struct 中最重要的两个属性 mmap 和 pgd。首先来看 pgd，pdg 中存放的是进行虚拟地址到物理地址转换的体系结构相关的页表首地址，ARM 体系结构中相应的就是一级页表的首地址。页表承担的任务就是将和进程相关的虚拟地址转换到物理地址，由于和进程相关所以放入 mm_struct 中进行管理是合理和必需的。物理内存管理已经涉及页表中内核地址空间的映射实现部分，用户空间的映射和其虚拟地址中存放数据的属性相关。由于各种数据属性的差异，属于个体属性，而转换页表属于整体属性，这也在形式上要求一个上层的结构管理这两部分属性。Linux 内核中这个上层的结构就是 mm_struct。管理进程中存放不同类型数据的虚拟空间管理结构就是 vm_area_struct，在 mm_struct 中由 mmap 对 vm_area_struct 统一进行管理。对于 vm_area_struct 详细的内容如下：

```
        struct vm_area_struct {
        struct mm_struct * vm_mm;    /* The address space we belong to. */
        unsigned long vm_start;      /* Our start address within vm_mm. */
        unsigned long vm_end;        /* The first byte after our end address within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        pgprot_t vm_page_prot;       /* Access permissions of this VMA. */
        unsigned long vm_flags;      /* Flags, see mm.h. */

        struct rb_node vm_rb;

        /*
         * For areas with an address space and backing store,
```

```
     * linkage into the address_space->i_mmap prio tree, or
     * linkage to the list of like vmas hanging off its node, or
     * linkage of vma in the address_space->i_mmap_nonlinear list.
     */
    union {
        struct {
            struct list_head list;
            void *parent;/* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
        } vm_set;

        struct raw_prio_tree_node prio_tree_node;
    } shared;

    /*
     * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
     * list, after a COW of one of the file pages. A MAP_SHARED vma
     * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
     * or brk vma(with NULL file)can only be in an anon_vma list.
     */
    struct list_head anon_vma_chain;/* Serialized by mmap_sem &
                          * page_table_lock */
    struct anon_vma *anon_vma;/* Serialized by page_table_lock */

    /* Function pointers to deal with this struct. */
    const struct vm_operations_struct *vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;        /* Offset(within vm_file)in PAGE_SIZE
                           units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;         /* File we map to(can be NULL). */
    void * vm_private_data;      /* was vm_pte(shared mem) */
    unsigned long vm_truncate_count;/* truncate_count or restart_addr */

#ifndef CONFIG_MMU
    struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
};
```

对于结构 vm_area_struct 几个比较重要的属性如下：

- vm_page_prot。用于相应虚拟空间页表的体系结构相关的属性。
- vm_flags。表示相应虚拟空间中数据的逻辑属性，如读、写、执行和增长以及操作方式等。
- vm_file。用于文件映射的虚拟空间，指向该区域映射的文件。它会和文件系统关联。
- vm_ops。所管理区域发生虚拟地址访问异常时相应操作的回调函数接口。内存管理主要是缺页异常操作接口。

整体的 Linux 内核中对用户进程虚拟空间的管理分布如图 4-40 所示。

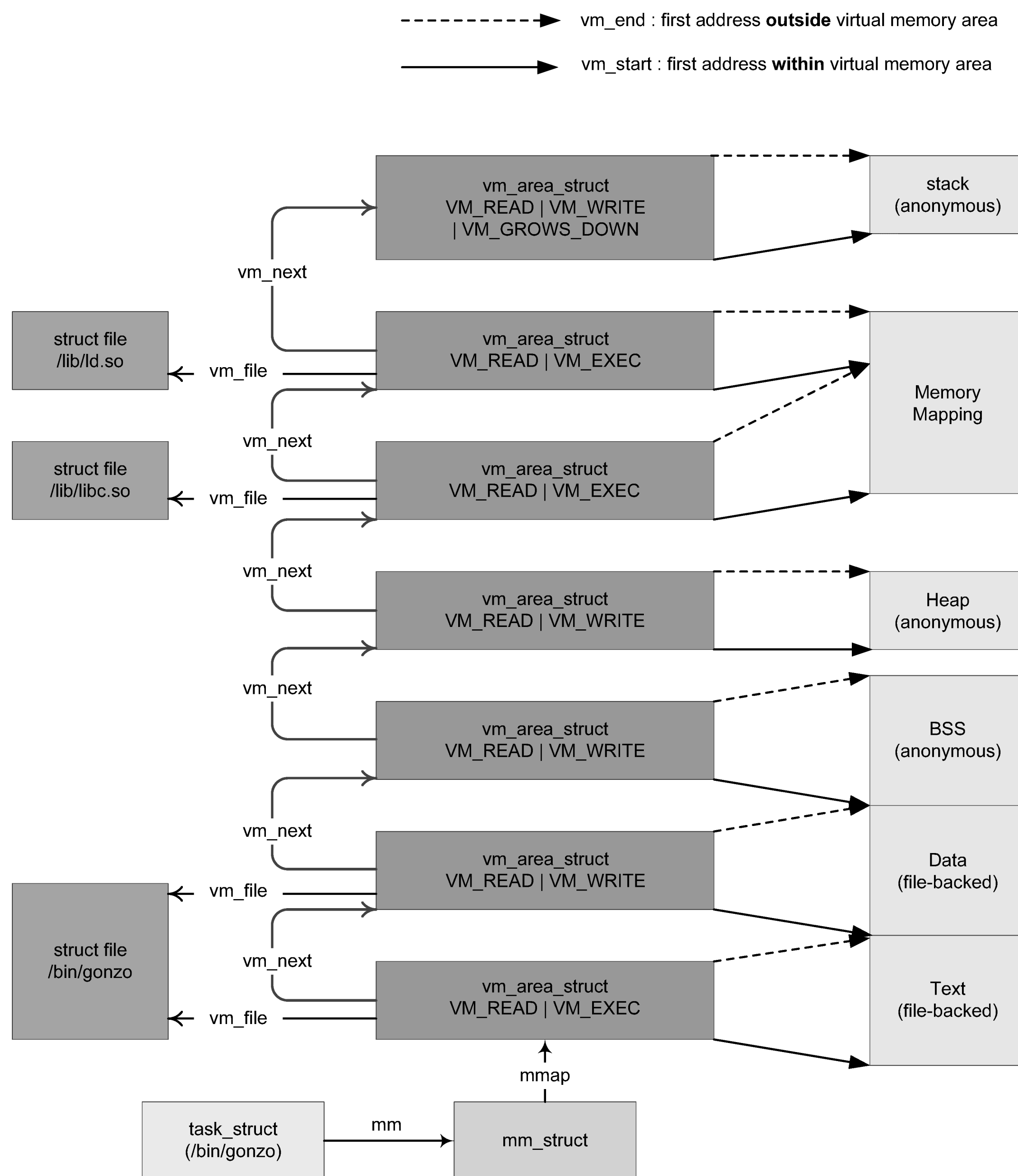


图 4-40　用户进程虚拟空间的管理分布

从图 4-40 中可见，可执行文件、各种类型虚拟空间以及文件系统的整体关系。这样对系统会有一个整体的理解。

再来看看对于虚拟地址映射的细节，如图 4-41 所示。

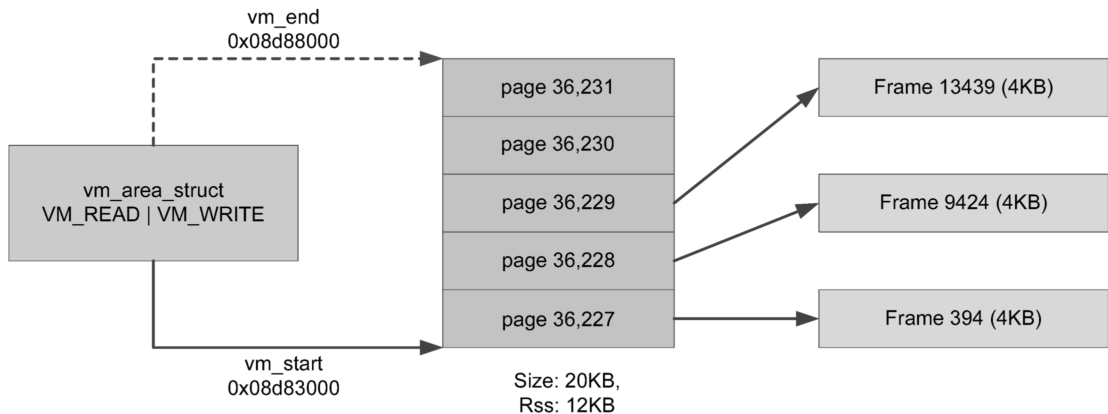


图 4-41　用户虚拟地址映射细节

从图 4-41 中可见，每个 vm_area_struct 管理的虚拟地址都会涉及相关的页表项，即相应虚拟地址范围内的页表项。而相应的页表项会有相关的访问属性，这些属性一般由 vm_area_struct 中的 vm_page_prot 来进行维护，在获得具体的物理页之后再与访问属性结合形成最终的页表项从而填入页表中完成最终的映射工作。对用户空间的物理页分配，Linux 内核同样采用将操作延迟到最后才执行，这样就会涉及地址访问异常，在异常处理中，内核会根据确认的用户空间虚拟地址及相应的进程找到合适的 vm_area_struct，其中的 vm_ops 会进行合适的操作来完成映射需要的工作。vm_ops 则会在 vm_area_struct 创建的时候根据相关的数据属性填入合适的操作接口。

注意 vm_area_struct 也为系统的扩展留下了很大的空间。进程中的虚拟地址空间是用户使用的直接接口，而虚拟空间中具体的映射内容可以是各种内容，可以根据需要进行扩展，这部分的扩展主要是在图 4-40 中的 mapping 部分体现。作为映射，内核中主要就是文件映射，相应的框架如图 4-42 所示。

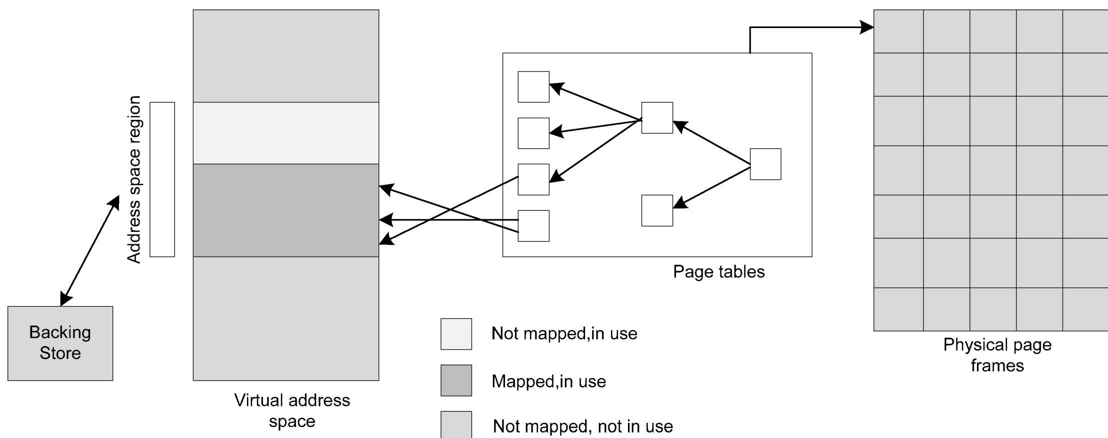


图 4-42　文件映射框架

图 4-42 中针对文件映射的虚拟地址空间由 address space 区域表示，为什么叫 address space？需要考虑的是对于文件来说也可以抽象为一种地址空间，而文件所在的地址空间相对于虚拟地址空间是完全不同的地址空间。Linux 内核对这种不同的地址空间抽象为 address_space 进行管理，address_space 的详细内容如下：

```
struct address_space {
    struct inode            *host;          /* owner: inode, block_device */
    struct radix_tree_root  page_tree;  /* radix tree of all pages */
    spinlock_t      tree_lock;/* and lock protecting it */
    unsigned int     i_mmap_writable;/* count VM_SHARED mappings */
    struct prio_tree_root    i_mmap;     /* tree of private and shared mappings */
    struct list_head   i_mmap_nonlinear;/*list VM_NONLINEAR mappings */
    spinlock_t        i_mmap_lock;/* protect tree, count, list */
    unsigned int      truncate_count;  /* Cover race condition with truncate */
    unsigned long     nrpages;/* number of total pages */
    pgoff_t           writeback_index;/* writeback starts here */
    const struct address_space_operations *a_ops;  /* methods */
    unsigned long     flags;     /* error bits/gfp mask */
    struct backing_dev_info *backing_dev_info;/* device readahead, etc */
    spinlock_t        private_lock;/* for use by the address_space */
    struct list_head  private_list;/* ditto */
    struct address_space      *assoc_mapping;/* ditto */
} __attribute__((aligned(sizeof(long))));
```

address_space 中比较重要的属性就是 a_ops，其中主要是进行文件映射的操作接口，相应的会将文件与页进行关联的操作。比如当文件映射的空间发生访问异常的时候，会通过 a_ops 相关的操作将文件合适的内容填入物理页，而当页内容需要回写到文件的时候也会通过相应的接口写回文件，可以说是文件和页之间的转换接口。在 i_mmap 可以找到所有与其相关的 vm_area_struct。而 vm_area_struct 可以通过其中的 vm_file 找到相应的 address_space，这样就将 address_space 和 vm_area_struct 完整地关联起来。

注意映射的内容是可以多个进程共享的，所以会有一个问题需要处理，需要知道哪些物理页面被不同的进程通过映射来使用了。进程映射的页可以直接通过查找页表确认，而如何通过页来查找使用的进程对内存管理来说也是很重要的，这是由于页换出时需要修改所有相关进程的页表。只要能够通过页来找到所有对应的 vm_area_struct 即可解决相关问题。先来看看对于物理页内存的管理实体 struct page，详细内容如下：

```
struct page {
    unsigned long flags;          /* Atomic flags, some possibly
                                   * updated asynchronously */
    atomic_t _count;              /* Usage count, see below. */
...
    union {
```

```
            struct {
            unsigned long private;/ *  Mapping - private opaque data:
    *  usually used for buffer_heads
    *  if PagePrivate set; used for
    *  swp_entry_t if PageSwapCache;
    *  indicates order in the buddy
    *  system if PG_buddy is set.
    * /
            struct address_space  * mapping;   / *  If low bit clear, points to
                                  *  inode address_space, or NULL.
                                  *  If page mapped as anonymous
                                  *  memory, low bit is set, and
                                  *  it points to anon_vma object:
                                  *  see PAGE_MAPPING_ANON below.
                                  * /
            };
    …
    };
```

从中看到了 struct address_space * mapping，这样加上之前 address_space 到 vm_area_struct 的通路，就可以通过物理页找到所有的 vm_area_struct，解决相应的问题。相对于文件映射还有一种是匿名映射，主要是图 4-40 中 anonymous 的部分。相应的反向查找也是通过 mapping 解决的，只是这里采用了小技巧，由于前面看到了对于 address_space 是要保证对齐的，所以低地址为 0，可以通过低地址进行属性标记，这里对匿名映射会进行标记并由高地址指向结构 anon_vma 来查找对应的 vm_area_struct，从而解决反向查找的问题。

4. 内存管理的重要参数

Linux 内核内存管理中，内核提供了相关的参数，供系统管理人员进行设置，以适用不同的系统并提高系统的效率。内存管理重要的设置参数都在/proc/sys/vm 目录下：

```
    - admin_reserve_kbytes
    - block_dump
    - compact_memory
    - dirty_background_bytes
    - dirty_background_ratio
    - dirty_bytes
    - dirty_expire_centisecs
    - dirty_ratio
    - dirty_writeback_centisecs
    - drop_caches
    - extfrag_threshold
    - hugepages_treat_as_movable
    - hugetlb_shm_group
```

- laptop_mode
- legacy_va_layout
- lowmem_reserve_ratio
- max_map_count
- memory_failure_early_kill
- memory_failure_recovery
- min_free_kbytes
- min_slab_ratio
- min_unmapped_ratio
- mmap_min_addr
- nr_hugepages
- nr_overcommit_hugepages
- nr_trim_pages　　　　(only if CONFIG_MMU = n)
- numa_zonelist_order
- oom_dump_tasks
- oom_kill_allocating_task
- overcommit_memory
- overcommit_ratio
- page - cluster
- panic_on_oom
- percpu_pagelist_fraction
- stat_interval
- swappiness
- user_reserve_kbytes
- vfs_cache_pressure
- zone_reclaim_mode

下面对其中的两个参数进行说明：

① /proc/sys/vm/lowmem_reserve_ratio，在之前已经介绍了对 Linux 内核，通常用户空间使用的内存会先从 high memory 获得，若没有才会在 low memory 中获取。主要原因在于内核重要的数据结构（基于 kmalloc/kmem_cache），都会从有限的 low memory 空间获取。这样可以知道 lowmem_reserve_ratio 比较适用于系统中有大块 high memory 的情况，若系统中只有 low memory 区域，或是 high memory 区域很有限，设定这个参数的意义就不会太大。若希望确保 low memory 的区域尽可能不要被能使用 high memory 的请求给分配走，就该把 lowmem_reserve_ratio 设定为 1(=100%)。若是设定的数值越高，则越有可能让 low memory 区域的内存分配给相应的请求使用。

② /proc/sys/vm/max_map_count，mmap 相应的应用程序地址空间之前已经进行了介绍，max_map_count 用以限定单一应用程序执行环境中最大的 mmap 的数量，默认值会等于 DEFAULT_MAX_MAP_COUNT。除非资源不足，否则默认值的配置已能符合目前应用程序的需求。

其他参数内核文档中有详细的说明。

至此，Linux 内核内存管理的主要内容都进行了介绍。

4.4.3 TI 芯片内存管理相关实现详解

1. 芯片片内内存使用

TI 芯片相关的内存管理，对片外内存来说，可以通过启动参数将需要内核管理的内存进行设置。而片内内存则有各种各样的使用方式，DM 3730 芯片是将其直接映射到内核虚拟地址空间的特定地址以供使用，采用该方法的原因主要是由于电源管理很多功能需要片内内存。这里介绍一下映射部分，具体的使用则在芯片电源管理部分进行介绍。相关映射代码如下：

```
static void __init omap_map_sram(void)
{
    unsigned long base;

    if(omap_sram_size == 0)
        return;

    if(cpu_is_omap34xx()) {
        /*
         * SRAM must be marked as non-cached on OMAP3 since the
         * CORE DPLL M2 divider change code(in SRAM)runs with the
         * SDRAM controller disabled, and if it is marked cached,
         * the ARM may attempt to write cache lines back to SDRAM
         * which will cause the system to hang.
         */
        omap_sram_io_desc[0].type = MT_MEMORY_NONCACHED;
    }

    omap_sram_io_desc[0].virtual = omap_sram_base;
    base = omap_sram_start;
    base = ROUND_DOWN(base, PAGE_SIZE);
    omap_sram_io_desc[0].pfn = __phys_to_pfn(base);
    omap_sram_io_desc[0].length = ROUND_DOWN(omap_sram_size, PAGE_SIZE);
    iotable_init(omap_sram_io_desc, ARRAY_SIZE(omap_sram_io_desc));

    printk(KERN_INFO "SRAM: Mapped pa 0x%08lx to va 0x%08lx size: 0x%lx\n",
    __pfn_to_phys(omap_sram_io_desc[0].pfn),
    omap_sram_io_desc[0].virtual,
        omap_sram_io_desc[0].length);

    /*
     * Normally devicemaps_init()would flush caches and tlb after
```

```
         * mdesc->map_io(), but since we're called from map_io(), we
         * must do it here.
         */
        local_flush_tlb_all();
        flush_cache_all();

        /*
         * Looks like we need to preserve some bootloader code at the
         * beginning of SRAM for jumping to flash for reboot to work...
         */
        memset((void *)omap_sram_base + SRAM_BOOTLOADER_SZ, 0,
            omap_sram_size - SRAM_BOOTLOADER_SZ);
    }
```

omap_map_sram 是将片内内存映射到内核的虚拟地址空间，使用的方式是按照 IO 映射进行的，主要是由于对于 ARM 体系结构使用统一的地址，另外需要关掉 cache 进行映射，映射到 omap_sram_base（使用变量是因为可以根据芯片调整）指定的地址。该接口是在板级支持的 map_io 中被调用，这样 DM 3730 内核就可以直接使用相应的虚拟地址进行操作了。使用相应空间的接口函数是 omap_sram_push，详细代码如下：

```
    void * omap_sram_push(void * start, unsigned long size)
    {
        if(size > (omap_sram_ceil - (omap_sram_base + SRAM_BOOTLOADER_SZ))){
            printk(KERN_ERR "Not enough space in SRAM\n");
            return NULL;
        }

        omap_sram_ceil -= size;
        omap_sram_ceil = ROUND_DOWN(omap_sram_ceil, sizeof(void *));
        memcpy((void *)omap_sram_ceil, start, size);
        flush_icache_range((unsigned long)omap_sram_ceil,
            (unsigned long)(omap_sram_ceil + size));

        return(void *)omap_sram_ceil;
    }
```

从 omap_sram_push 可见，其采用从后往前的方式进行分配，主要是将需要的数据复制到相应的片内空间。芯片电源管理部分可知具体的数据内容，这里可以先说明一下，其内容主要是执行的代码。

2. 芯片保留物理内存

SoC 内部通常有些模块需要在启动时保留部分内存空间。这样做的原因是由于需要连续的物理内存空间，并且对内核的物理内存管理影响最小。DM 3730 同样有该需求，看看 DM

3730 是如何实现相应功能的：

```
//该接口是给 machine_desc 提供的 board 板级 memory reserve 的接口
//其中 vram_reserve 通过 bootargs 这种初始化 vram 参数的方式保留空间
//dsp_reserve 则是为了 dsp_bridge 保留空间.
void __init omap_reserve(void)
{
    omapfb_reserve_sdram_memblock();
    omap_vram_reserve_sdram_memblock();
    omap_dsp_reserve_sdram_memblock();
}
```

之前在介绍 Linux 内核初始化时看到 arm_memblock_init，其中会根据启动时的参数获得物理内存的 memblock 信息，在该接口中会调用板级的 reserve 接口，就可以保留内存空间，使得 Linux 内核不管理这部分物理内存，而是留给板级相关的驱动去管理，这就相当于 Linux 内核开出一块保留区域给处理器自由使用。DM 3730 就是利用该接口，保留了相关的内存，为特殊的驱动使用，包括显示驱动和 DSP 相关功能部分。具体的内容还是在实际驱动中进行介绍。

3. 独立 cmem 管理

之前简单说明芯片厂商的独立内存管理，针对于 TI 的视频芯片，TI 提供了 cmem 的内存管理驱动，主要是连续内存管理。其主要目的是针对视频 SoC 的视频加速应用，如视频编解码功能。视频编解码功能通常属于应用的范畴，而如果通过硬件加速来实现视频编解码，则其需要连续的物理空间，这就相当于要在应用中使用连续的物理内存。如果由 Linux 内核直接管理和提供这部分内存是无法实现的，这样芯片厂商就开发相应的驱动，通过将物理内存交由驱动管理，而不是内核直接管理来实现该功能。具体的映射则通过虚拟地址的映射机制来实现。

对于 TI 的 cmem 有两种分配模式：一种是采用 pool 算法实现的，通过参数来设定所管理的物理空间的地址范围及每个内存池的大小及数目，以驱动的形式管理这部分内存；另一种是 heap 模式，把内存区域作为堆来使用。这两种模式对于内存的管理都是通过 IO 控制来完成。下面是一个使用的例子：

```
insmod cmemk.ko phys_start=0x83200000 phys_end=0x88000000 allowOverlap=1 phys_start_1=
0x00001000 phys_end_1=0x00008000 pools_1=1x28672
```

这个例子就是对大块的物理内存使用堆模式，而对于片内的内存采用了 pool 模式管理。

当然对内存管理不可忽视的是处理器 cache 相关的操作，为了给相关的应用提供完整的功能，独立的内存管理同样也要提供 cache 相关的操作接口，对于 cmem 提供的完整操作接口如下：

```
#define CMEM_IOCALLOC              1
#define CMEM_IOCALLOCHEAP          2
```

```
#define CMEM_IOCFREE                    3
#define CMEM_IOCGETPHYS                 4
#define CMEM_IOCGETSIZE                 5
#define CMEM_IOCGETPOOL                 6
#define CMEM_IOCCACHE                   7
#define CMEM_IOCGETVERSION              8
#define CMEM_IOCGETBLOCK                9
#define CMEM_IOCREGUSER                 10
#define CMEM_IOCGETNUMBLOCKS            11
#defineCMEM_IOCCACHEWBINV    CMEM_IOCCACHE | CMEM_WB | CMEM_INV
#define CMEM_IOCCACHEWB                 CMEM_IOCCACHE | CMEM_WB
#define CMEM_IOCCACHEINV                CMEM_IOCCACHE | CMEM_INV
#define CMEM_IOCALLOCCACHED             CMEM_IOCALLOC | CMEM_CACHED
#define CMEM_IOCALLOCHEAPCACHED   CMEM_IOCALLOCHEAP | CMEM_CACHED
#define CMEM_IOCFREEHEAP                CMEM_IOCFREE | CMEM_HEAP
#define CMEM_IOCFREEPHYS                CMEM_IOCFREE | CMEM_PHYS
#define CMEM_IOCFREEHEAPPHYS     CMEM_IOCFREE | CMEM_HEAP | CMEM_PHYS
```

从功能看已经设计了完整的内存管理和 cache 管理功能。有了该内存管理功能，TI 的多媒体框架和编解码库就可以使用连续的物理内存空间，高效地完成多媒体处理相关的功能。

4.5 直接存储器访问单元（DMA）

从硬件的角度，DMA 可以说是继中断之后另一个重要的里程碑。它开创了硬件加速的时代来解放处理器资源，继 DMA 之后各种特定功能的硬件加速应运而生，但只有 DMA 有如此广泛的应用。现如今应用处理器都会集成 DMA，可见 DMA 的重要性。

4.5.1 DMA 使用和管理基本需求

要了解 DMA 使用和管理的基本需求，需要清楚 DMA 在系统框架中的位置，图 4-43 是带有 DMA 的系统框图。

在图 4-43 中可见 DMA 是在总线上，它可以访问 memory 和外设，另外 CPU 可以访问控制 DMA，外设也可以通过信号来发起 DMA 的操作，只有 memory 是纯的数据设备。DMA 可以读写 memory 和外设，这样就可以在外设和内存之间传送数据，注意这里有方向问题。另外也可通过 DMA 在 memory 之间传送数据，这样就有复制和逻辑转换的功能。从功能的角度，需要 DMA 在能力范围内完成各种数据传输的工作，而对内存的组织尽量有更广泛的适用范围，还需要保证数据传输的正确性和一致性，以及与 CPU 访问的同步。从管理的角度考虑，DMA 是一种公共资源，公共资源的管理需要能够随时请求、释放和设置，并且需要保证在使用资源者之间不发生冲突。由于 DMA 在使用的时候离不开 memory 资源，所以 DMA 的使用还要包括对其使用的内存资源的管理。

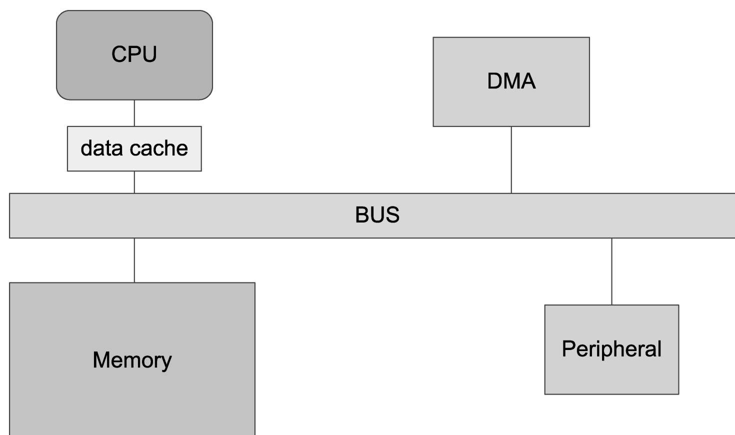


图 4-43　带 DMA 系统框图

4.5.2　DMA 使用和管理框架介绍

由于 DMA 和芯片体系相关，各个芯片厂商的设计接口参数是不同的。由于 DMA 需要访问内存，所以在早期的 Linux 内核提供的使用和管理框架主要是内存管理的框架，这些框架用于分配和管理 DMA 要使用的内存资源，规范 DMA 和 CPU 访问内存区域的流程。所有这些管理方式都在 dma-mapping.h 中有声明。这里用 mapping 的概念表示内存资源的管理也是合适的，毕竟这些内存无论从 CPU 还是 DMA 访问都相当于一个映射的过程。

1. 一致性 DMA 映射管理

首先常用的是一致性 DMA 映射，这里的一致性主要是指 CPU 和 DMA 访问的一致性，其中相应的内存区域一般都是关闭 cache 的，这样数据可以保持一致性。另外相应的区域 CPU 和设备是可以同时访问的，这是建立在一致性基础上的。对系统来说，该特性还是比较重要的，这样减少了同步的操作。

Linux 内核提供了一组 DMA 一致映射的接口，具体如下：

```
/**
 * dma_alloc_coherent - allocate consistent memory for DMA
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @size: required memory size
 * @handle: bus-specific DMA address
 *
 * Allocate some uncached, unbuffered memory for a device for
 * performing DMA.  This function allocates pages, and will
 * return the CPU-viewed address, and sets @handle to be the
 * device-viewed address.
 */
extern void *dma_alloc_coherent(struct device *, size_t, dma_addr_t *, gfp_t);
```

```
/**
 * dma_free_coherent - free memory allocated by dma_alloc_coherent
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @size: size of memory originally requested in dma_alloc_coherent
 * @cpu_addr: CPU-view address returned from dma_alloc_coherent
 * @handle: device-view address returned from dma_alloc_coherent
 *
 * Free(and unmap)a DMA buffer previously allocated by
 * dma_alloc_coherent().
 *
 * References to memory and mappings associated with cpu_addr/handle
 * during and after this call executing are illegal.
 */
extern void dma_free_coherent(struct device *, size_t, void *, dma_addr_t);

/**
 * dma_mmap_coherent - map a coherent DMA allocation into user space
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @vma: vm_area_struct describing requested user mapping
 * @cpu_addr: kernel CPU-view address returned from dma_alloc_coherent
 * @handle: device-view address returned from dma_alloc_coherent
 * @size: size of memory originally requested in dma_alloc_coherent
 *
 * Map a coherent DMA buffer previously allocated by dma_alloc_coherent
 * into user space.  The coherent DMA buffer must not be freed by the
 * driver until the user space mapping has been released.
 */
int dma_mmap_coherent(struct device *, struct vm_area_struct *,
        void *, dma_addr_t, size_t);
```

这里保留相应的注释，主要是相应的注释都很明确地表述了相应接口的功能。对这些接口的使用，通常是在要使用 DMA 的设备驱动初始化时分配相应的空间，在驱动退出时释放相应的空间。如果驱动支持在相应空间被应用程序访问，则通过 dma_mmap_coherent 来进行映射。需要注意的是，特殊的设备在映射时，相同的物理内存空间对于 CPU 和 DMA 是不同的地址，相应的分配接口是通过传址参数来返回 DMA 视角的地址，通过返回值返回 CPU 视角的地址，这样在系统级别达到一致性。

一致性映射的空间在整个驱动的生命周期中都被使用，可以说是驱动的一个组成部分，所以是可以对应用层开放并提供映射接口的。

一致性接口的粒度比较大，是以页为单位的，并且不能提供大量的 DMA 传输需要的空间。

参考 kmem_cache 的方式，DMA 也可以将所有的一定大小的空间需求组织在一起，进行

整体的物理内存分配，而该大小的单个 DMA 分配需求，作为细粒度的 DMA 操作，通过相应的接口在整体申请的 DMA 空间中进行分配和操作。这样就形成一个池的功能，而相应的接口就是 dma_pool。详细的说明如下：

```
/**
 * dma_pool_create - Creates a pool of consistent memory blocks, for dma.
 * @name: name of pool, for diagnostics
 * @dev: device that will be doing the DMA
 * @size: size of the blocks in this pool.
 * @align: alignment requirement for blocks; must be a power of two
 * @boundary: returned blocks won't cross this power of two boundary
 * Context: !in_interrupt()
 *
 * Returns a dma allocation pool with the requested characteristics, or
 * null if one can't be created.  Given one of these pools, dma_pool_alloc()
 * may be used to allocate memory.  Such memory will all have "consistent"
 * DMA mappings, accessible by the device and its driver without using
 * cache flushing primitives.  The actual size of blocks allocated may be
 * larger than requested because of alignment.
 *
 * If @boundary is nonzero, objects returned from dma_pool_alloc() won't
 * cross that size boundary.  This is useful for devices which have
 * addressing restrictions on individual DMA transfers, such as not crossing
 * boundaries of 4KBytes.
 */
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                 size_t size, size_t align, size_t boundary)

/**
 * dma_pool_alloc - get a block of consistent memory
 * @pool: dma pool that will produce the block
 * @mem_flags: GFP_* bitmask
 * @handle: pointer to dma address of block
 *
 * This returns the kernel virtual address of a currently unused block,
 * and reports its dma address through the handle.
 * If such a memory block can't be allocated, %NULL is returned.
 */
void *dma_pool_alloc(struct dma_pool *pool, gfp_t mem_flags,
                     dma_addr_t *handle)

/**
 * dma_pool_free - put block back into dma pool
```

```
 * @pool: the dma pool holding the block
 * @vaddr: virtual address of block
 * @dma: dma address of block
 *
 * Caller promises neither device nor driver will again touch this block
 * unless it is first re-allocated.
 */
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t dma)

/**
 * dma_pool_destroy - destroys a pool of dma memory blocks.
 * @pool: dma pool that will be destroyed
 * Context: !in_interrupt()
 *
 * Caller guarantees that no more memory from the pool is in use,
 * and that nothing will try to use the pool after this call.
 */
void dma_pool_destroy(struct dma_pool *pool)
```

整体的 dma pool 创建和销毁可使用 dma_pool_create 和 dma_pool_destroy，而细粒度的分配和释放可使用 dma_pool_alloc 和 dma_pool_free。

2. 流式 DMA 映射管理

一致性映射是消耗比较大的映射，主要是一旦分配后只能在驱动退出时才能释放，这样相当于驱动一直占用内存资源，而并不管是否要进行 DMA 相关的操作。由于驱动并不是一直都需要使用相应的内存资源进行 DMA 操作，这在一定程度上造成了比较大的开销。为了解决该问题，Linux 内核提供了流式 DMA 映射方式，相应的内存只有在需要进行 DMA 操作之前才进行 DMA 相关的映射，DMA 操作完之后进行映射的释放，这样在进行 DMA 操作之外的时间由 CPU 进行处理。这样保证相应的空间，只在 DMA 操作期间才归驱动的 DMA 操作所有，之外 CPU 可以进行任何处理，释放修改等都可以，比较适合用户空间通过驱动传输数据的操作。相应的接口细节如下：

```
/**
 * dma_map_single - map a single buffer for streaming DMA
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @cpu_addr: CPU direct mapped address of buffer
 * @size: size of buffer to map
 * @dir: DMA transfer direction
 *
 * Ensure that any data held in the cache is appropriately discarded
 * or written back.
 *
```

```
 * The device owns this memory once this call has completed.  The CPU
 * can regain ownership by calling dma_unmap_single() or
 * dma_sync_single_for_cpu().
 */
static inline dma_addr_t dma_map_single(struct device *dev, void *cpu_addr,
        size_t size, enum dma_data_direction dir)

/**
 * dma_map_page - map a portion of a page for streaming DMA
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @page: page that buffer resides in
 * @offset: offset into page for start of buffer
 * @size: size of buffer to map
 * @dir: DMA transfer direction
 *
 * Ensure that any data held in the cache is appropriately discarded
 * or written back.
 *
 * The device owns this memory once this call has completed.  The CPU
 * can regain ownership by calling dma_unmap_page().
 */
static inline dma_addr_t dma_map_page(struct device *dev, struct page *page,
             unsigned long offset, size_t size, enum dma_data_direction dir)

/**
 * dma_unmap_single - unmap a single buffer previously mapped
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @handle: DMA address of buffer
 * @size: size of buffer(same as passed to dma_map_single)
 * @dir: DMA transfer direction(same as passed to dma_map_single)
 *
 * Unmap a single streaming mode DMA translation.  The handle and size
 * must match what was provided in the previous dma_map_single() call.
 * All other usages are undefined.
 *
 * After this call, reads by the CPU to the buffer are guaranteed to see
 * whatever the device wrote there.
 */
static inline void dma_unmap_single(struct device *dev, dma_addr_t handle,
        size_t size, enum dma_data_direction dir)

/**
```

```
 * dma_unmap_page - unmap a buffer previously mapped through dma_map_page( )
 * @ dev: valid struct device pointer, or NULL for ISA and EISA - like devices
 * @ handle: DMA address of buffer
 * @ size: size of buffer( same as passed to dma_map_page)
 * @ dir: DMA transfer direction( same as passed to dma_map_page)
 *
 * Unmap a page streaming mode DMA translation.  The handle and size
 * must match what was provided in the previous dma_map_page( ) call.
 * All other usages are undefined.
 *
 * After this call, reads by the CPU to the buffer are guaranteed to see
 * whatever the device wrote there.
 */
static inline void dma_unmap_page( struct device *dev, dma_addr_t handle,
        size_t size, enum dma_data_direction dir)
```

可见这些接口有一段数据的映射也有物理页的映射，这里接口并不负责分配空间，而是解决内存所有者的问题，并附加进行方向的检查，保证数据的一致性以及和 CPU 的同步。

3. 散列式 DMA 映射管理

流式 DMA 映射处理器和 DMA 一次只能交互单次的 DMA 传输数据，相对来说效率低一些。在内核中很多模块及层次都是需要批量处理数据的能力，这就要求进行 DMA 相关映射管理的时候，要能进行数据批量的交互，批量交给驱动进行 DMA 传输工作，驱动接收到批量数据后再由 DMA 进行循环操作完成批量传输。相关的接口细节如下：

```
/**
 * dma_map_sg - map a set of SG buffers for streaming mode DMA
 * @ dev: valid struct device pointer, or NULL for ISA and EISA - like devices
 * @ sg: list of buffers
 * @ nents: number of buffers to map
 * @ dir: DMA transfer direction
 *
 * Map a set of buffers described by scatterlist in streaming mode for DMA.
 * This is the scatter - gather version of the dma_map_single interface.
 * Here the scatter gather list elements are each tagged with the
 * appropriate dma address and length.  They are obtained via
 * sg_dma_{address,length}.
 *
 * Device ownership issues as mentioned for dma_map_single are the same
 * here.
 */
int dma_map_sg( struct device *dev, struct scatterlist *sg, int nents,
        enum dma_data_direction dir)
```

```
/**
 * dma_unmap_sg - unmap a set of SG buffers mapped by dma_map_sg
 * @dev: valid struct device pointer, or NULL for ISA and EISA - like devices
 * @sg: list of buffers
 * @nents: number of buffers to unmap(returned from dma_map_sg)
 * @dir: DMA transfer direction(same as was passed to dma_map_sg)
 *
 * Unmap a set of streaming mode DMA translations.  Again, CPU access
 * rules concerning calls here are the same as for dma_unmap_single().
 */
void dma_unmap_sg(struct device *dev, struct scatterlist *sg, int nents,
        enum dma_data_direction dir)
```

相应的批量映射的区域由参数 sg 表示，在进行 DMA 操作之前，将 dma_map_sg 映射给驱动，驱动批量 DMA 操作结束后，调用 dma_unmap_sg，将相关区域归还，上层框架会继续处理相关区域。

4. 体系结构相关接口

在体系结构中，具体的设备允许通过 DMA 访问的地址空间可能是受限的，这就需要对相应的设备进行标注，从而在上面的映射接口中进行正确的操作。相应的限制会和设备绑定，因为 DMA 的最终使用者会以物理设备和逻辑设备的形式存在，这样的限制和设备绑定更合理。相应的详细接口如下：

```
/*
 * Return whether the given device DMA address mask can be supported
 * properly.  For example, if your device can only drive the low 24 - bits
 * during bus mastering, then you would pass 0x00ffffff as the mask
 * to this function.
 *
 * FIXME: This should really be a platform specific issue - we should
 * return false if GFP_DMA allocations may not satisfy the supplied 'mask'.
 */
static inline int dma_supported(struct device *dev, u64 mask)

static inline int dma_set_mask(struct device *dev, u64 dma_mask)
```

5. DMA engine 框架

以上都是和 DMA 操作的内存管理相关的框架和接口，并没有涉及 DMA 操作的统一接口和 DMA 自身资源管理部分。有统一的接口可以使得驱动的抽象程度更高，减少不同芯片相同功能驱动的差异，提高可移植性，可以提供统一的 IP 级别的驱动。但是这在早期的 Linux 内核中都没有提供，新的 Linux 内核才通过 DMA engine 框架提供了该部分功能。DMA engine 是上层框架，主要是对 DMA 的操作和属性进行管理和抽象，其中将 DMA 作为设备来

管理。要实现完整的功能还需要各种芯片的支持以及各种设备驱动的移植。由于是新内核才提供的功能，并不是所有的芯片都支持该功能，所以下面只是对接口进行简单的说明。

分配 DMA 内部 channel 的接口：

```
struct dma_chan  *dma_request_channel(dma_cap_mask_t mask,
                        dma_filter_fn filter_fn,
                        void  *filter_param);
```

DMA 特殊属性配置的接口：

```
int dmaengine_slave_config(struct dma_chan  *chan,
                           struct dma_slave_config  *config)
```

DMA 传输的描述结构是 struct dma_async_tx_descriptor。

设备中会有如下的接口来绑定物理内存和设备的 DMA 操作，并返回抽象的 DMA 传输描述结构：

```
struct dma_async_tx_descriptor  *( *chan->device->device_prep_slave_sg)(
        struct dma_chan  *chan, struct scatterlist  *sgl,
        unsigned int sg_len, enum dma_data_direction direction,
        unsigned long flags);
struct dma_async_tx_descriptor  *( *chan->device->device_prep_dma_cyclic)(
        struct dma_chan  *chan, dma_addr_t buf_addr, size_t buf_len,
        size_t period_len, enum dma_data_direction direction);
struct dma_async_tx_descriptor  *( *device_prep_interleaved_dma)(
        struct dma_chan  *chan, struct dma_interleaved_template  *xt,
        unsigned long flags);
```

发起传输的接口：

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor  *desc)
```

另外还有 DMA 传输的控制接口：

```
int dmaengine_terminate_all(struct dma_chan  *chan)
int dmaengine_pause(struct dma_chan  *chan)
int dmaengine_resume(struct dma_chan  *chan)
enum dma_status dma_async_is_tx_complete(struct dma_chan  *chan,
          dma_cookie_t cookie, dma_cookie_t  *last, dma_cookie_t  *used)
```

以上接口可以说已经抽象出各种 DMA 操作及控制，是很好的上层框架。

4.5.3 TI 芯片 DMA 使用和管理相关实现详解

要了解具体芯片的 DMA 使用和管理，首先要看一下硬件的设计。DM 3730 芯片内部的

DMA 是 System DMA（SDMA）。SDMA 和其他模块联系的系统框图如图 4-44 所示。图 4-44 引自《DM 3730 芯片手册》中第 2339 页的框图。

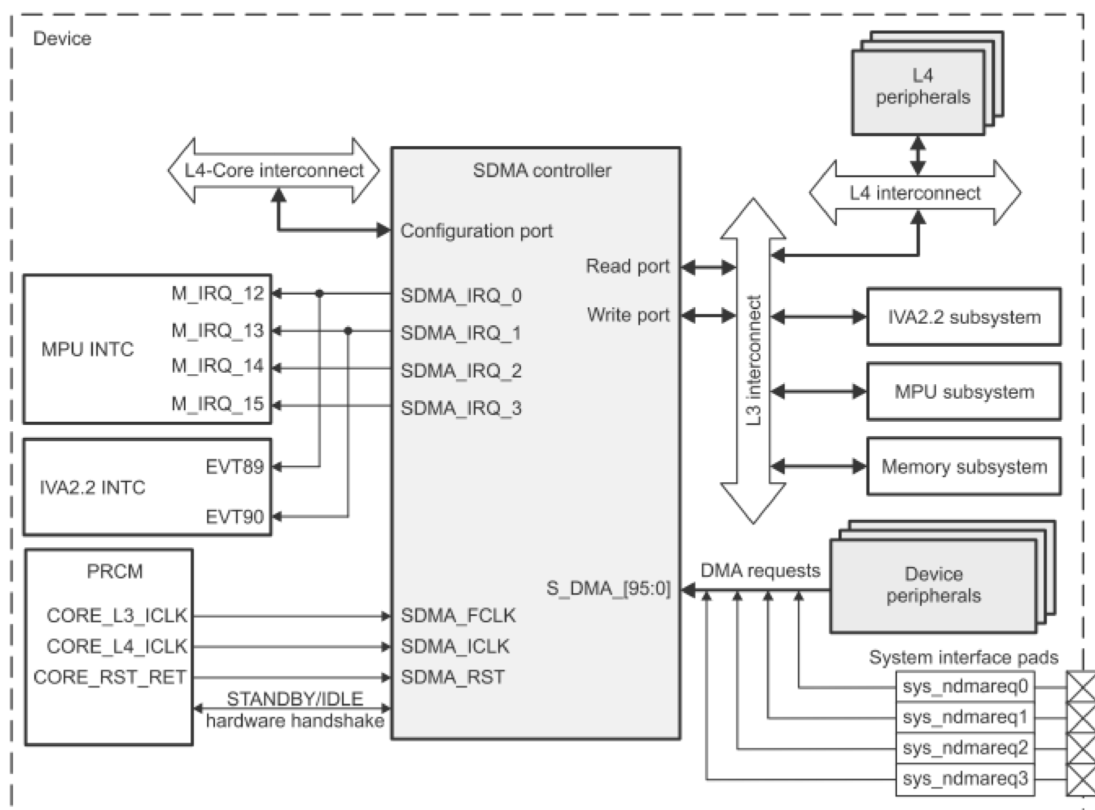


图 4-44　DM 3730 SDMA 与其他模块联系系统框图

从图 4-44 可见，SDMA 共有 96 个 DMA 请求，用于设备请求 DMA 进行数据传输处理。SDMA 可以访问到外设并可以响应芯片外部的 DMA 请求。为了提高整个的 DMA 性能，DMA 提供多个中断信号给处理器，来加速系统对 DMA 处理数据过程的响应。需要注意的是 DM 3730 中有两个主要的处理器：一个是 ARM MPU，另一个是 IVA（实际是 DSP）。两个主核都有对 DMA 的需求，所以在 DMA 处理部分也要考虑多核相关的支持，保证多核间不会对 DMA 内部资源访问造成冲突。

SDMA 系统内部的框架如图 4-45 所示。图 4-45 引自《DM 3730 芯片手册》中第 2344 页框图。

从图 4-45 可见，SDMA 只有一个用于读操作和一个用于写操作的端口，SDMA 内部通过 logical channel 管理不同的传输请求，通过调度器以及优先级来进行调度，另外通过 FIFO 来缓冲每个逻辑 channel 的数据。SDMA 有 32 个逻辑 channel，每个 channel 均可单独设置，并且可以将多个 channel 串成逻辑链。

接下来分几部分进行详细的解析。

1. 逻辑 channel 的管理

对于逻辑 channel 的管理主要是请求和释放。

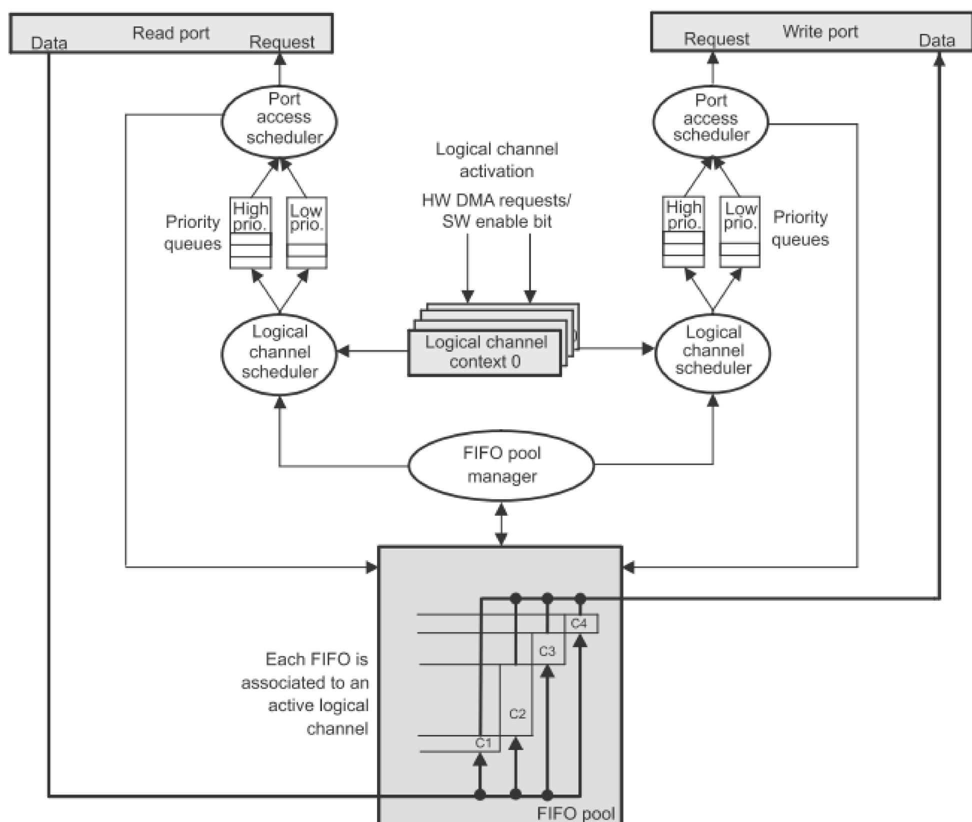


图 4-45　SDMA 系统内部框图

```
//其他模块请求 DMA 的接口函数,要设置 DMA channel 是哪个设备使用
int omap_request_dma(int dev_id, const char * dev_name,
                void( * callback)(int lch, u16 ch_status, void * data),
                void * data, int * dma_ch_out)
{
    int ch, free_ch = -1;
    unsigned long flags;
    struct omap_dma_lch * chan;

    //中断加锁保护 channel 管理的结构
    spin_lock_irqsave(&dma_chan_lock, flags);

    //查找没有使用的 channel 对应的管理结构
    for(ch =0; ch < dma_chan_count; ch ++ ){
        if(free_ch == -1 && dma_chan[ch]. dev_id == -1){
            free_ch = ch;
            if(dev_id ==0)
                break;
```

```
            }
        }
        if(free_ch == -1){
            spin_unlock_irqrestore(&dma_chan_lock, flags);
            return -EBUSY;
        }

        //找到相应的管理结构
        //设置相应的 device_id
        chan = dma_chan + free_ch;
        chan->dev_id = dev_id;

        //如果有 channel 清除寄存器操作则清除,omap3 中没有该操作
        if(p->clear_lch_regs)
            p->clear_lch_regs(free_ch);

        //omap3 有单独的 clear 操作,omap3 中 channel 相关的寄存器写 0
        //清除状态 csr 以便可以 start_dma
        if(cpu_class_is_omap2())
            omap_clear_dma(free_ch);

        //可以释放锁
        spin_unlock_irqrestore(&dma_chan_lock, flags);

        //设置其他属性
        chan->dev_name = dev_name;
        //callback 及 data 为 DMA 操作中断 irq 处理函数的回调及所用参数
        chan->callback = callback;
        chan->data = data;
        chan->flags = 0;

#ifndef CONFIG_ARCH_OMAP1
        if(cpu_class_is_omap2()){
            //目前没有在 chain 中也没有 next_link,相应的域要初始化
            chan->chain_id = -1;
            chan->next_linked_ch = -1;
        }
#endif

        //记录要使能的中断包括 dma_drop(丢失 DMA 中断)
        //和 block_irq(block 传输完毕中断)
        chan->enabled_irqs = OMAP_DMA_DROP_IRQ | OMAP_DMA_BLOCK_IRQ;
```

```
    //omap3 还要使能 MISALIGNED_ERR_IRQ(地址没有对齐错误中断)
    //和 TRANS_ERR_IRQ(传输错误中断)以便检查传输错误
    if(cpu_class_is_omap1())
        chan->enabled_irqs |= OMAP1_DMA_TOUT_IRQ;
    else if(cpu_class_is_omap2())
        chan->enabled_irqs |= OMAP2_DMA_MISALIGNED_ERR_IRQ | OMAP2_DMA_TRANS_
ERR_IRQ;
...
    if(cpu_class_is_omap2()){
        //enable channel 对 MPU 的中断
        omap2_enable_irq_lch(free_ch);
        //enable 相应的 channel 的中断,写 channel 的 cicr 寄存器
        omap_enable_channel_irq(free_ch);
        /* Clear the CSR register and IRQ status register */
        //清除 channel 的 csr 中断寄存器
        p->dma_write(OMAP2_DMA_CSR_CLEAR_MASK, CSR, free_ch);
        //清除 channel 对 MPU 的 irqstatus 相应位,以便中断可用
        p->dma_write(1 << free_ch, IRQSTATUS_L0, 0);
    }

    //返回相应的 channel 号
    *dma_ch_out = free_ch;

    return 0;
}

//释放相应的 DMA channel
void omap_free_dma(int lch)
{
    unsigned long flags;

    if(dma_chan[lch].dev_id == -1){
        pr_err("omap_dma: trying to free unallocated DMA channel %d\n",lch);
        return;
    }
    ...
    if(cpu_class_is_omap2()){
        //首先要 disable 相应的 channel 对 MPU 的中断
        omap2_disable_irq_lch(lch);

        /* Clear the CSR register and IRQ status register */
```

```
            //清除 channel 中断状态寄存器
            p->dma_write(OMAP2_DMA_CSR_CLEAR_MASK, CSR, lch);
            //清除 channel 对 MPU 的 irq status
            p->dma_write(1 << lch, IRQSTATUS_L0, lch);

            /* Disable all DMA interrupts for the channel. */
            //disable channel 中断写 cicr 为 0,关掉所有中断
            p->dma_write(0, CICR, lch);

            /* Make sure the DMA transfer is stopped. */
            //写 channel control register 为 0,停掉 DMA
            p->dma_write(0, CCR, lch);
            //清除 channel 的寄存器
            omap_clear_dma(lch);
        }

        //清除其 channel 管理实体相应的属性,以便可以再请求
        spin_lock_irqsave(&dma_chan_lock, flags);
        dma_chan[lch].dev_id = -1;
        dma_chan[lch].next_lch = -1;
        dma_chan[lch].callback = NULL;
        spin_unlock_irqrestore(&dma_chan_lock, flags);
    }
```

2. 逻辑 channel 的参数设置接口

逻辑 channel 的各种设置包括很多的寄存器操作，会涉及芯片手册的一些细节，对细节中设计芯片手册的内容，直接在注释中以英文的形式存在。相应的接口主要是对申请到的逻辑 channel 进行设置：

```
    //设置 channel 的优先级,对于 omap3 sdma 只有 high/low 两个优先级,主要差别是分
    //得 fifo 不同。omap3 上设置 ccr 的 read 优先级实现
    void omap_set_dma_priority(int lch, int dst_port, int priority)
    {
        unsigned long reg;
        u32 l;
    ...
        //omap3 上主要是设置 ccr 的 read 优先级
        if(cpu_class_is_omap2()){
            u32 ccr;

            ccr = p->dma_read(CCR, lch);
            if(priority)
```

```
            ccr |= (1 << 6);
        else
            ccr &= ~(1 << 6);
        p->dma_write(ccr, CCR, lch);
    }
}

//设置 channel 的 transfer 属性,主要是 ccr 的设置
void omap_set_dma_transfer_params(int lch, int data_type, int elem_count,
                    int frame_count, int sync_mode,
                    int dma_trigger, int src_or_dst_synch)
{
    u32 l;

    //设置 data_type 是在 cdsp 低 2bits
    //#define OMAP_DMA_DATA_TYPE_S8      0x00
    //#define OMAP_DMA_DATA_TYPE_S16     0x01
    //#define OMAP_DMA_DATA_TYPE_S32     0x02
    l = p->dma_read(CSDP, lch);
    l &= ~0x03;
    l |= data_type;
    p->dma_write(l, CSDP, lch);
...
    if(cpu_class_is_omap2() && dma_trigger) {
        u32 val;

        val = p->dma_read(CCR, lch);

        /* DMA_SYNCHRO_CONTROL_UPPER depends on the channel number */
        //默认关掉 prefetch 等,关联的硬件 dma_request 信号由 dma_trigger 表示,通常为 0 ~
        //4 bit,5 个 bit 表示低 5 bits,另外 19 ~ 20 bit 表示高 2 bits。如果是 software synchronization
        //trigger 的 DMA 操作,dma_trigger 应该设置为 0
        val &= ~((1 << 23) | (3 << 19) | 0x1f);
        val |= (dma_trigger & ~0x1f) << 14;
        val |= dma_trigger & 0x1f;

// Frame synchronization
// This bit used with the BS to see how the DMA request is serviced in a synchronized transfer
// FS = 0 and BS = 0: An element is transferred once a DMA request is made.
// FS = 0 and BS = 1: An entire block is transferred once a DMA request is made.
// FS = 1 and BS = 0: An entire frame is transferred once a DMA request is made.
// FS = 1 and BS = 1: A packet is transferred once a DMA request is made.
```

```
// All these different transfers can be interleaved on the port with other DMA requests.
//注意 packet 同步模式要求 fs = 1 并且 bs = 1,所以宏定义中
// sync_packet 为 3 包括了 sync_frame 和 sync_block
        if(sync_mode & OMAP_DMA_SYNC_FRAME)
            val |= 1 << 5;
        else
            val &= ~(1 << 5);

        if(sync_mode & OMAP_DMA_SYNC_BLOCK)
            val |= 1 << 18;
        else
            val &= ~(1 << 18);

        //Prefetch mode is active only when destination is synchronized
        //设置源还是目的 dma_request 来触发 dma channel 的传输
        if(src_or_dst_synch == OMAP_DMA_DST_SYNC_PREFETCH){
            val &= ~(1 << 24);/* dest synch */
            val |= (1 << 23);/* Prefetch */
        } else if(src_or_dst_synch){
            val |= 1 << 24;/* source synch */
        } else {
            val &= ~(1 << 24);/* dest synch */
        }
        p->dma_write(val, CCR, lch);
    }

    //设置传输的 count,包括一帧的 element 数和一个 block 的 frame 数
    p->dma_write(elem_count, CEN, lch);
    p->dma_write(frame_count, CFN, lch);
}

//对 source 和 destination 的参数设置物理上有寄存器 csdp 即 channel source destination parameter
//反映
//设置 source 的参数,omap3 中没有 src_port 参数

//src_amode 的设置在 ccr 寄存器中
//Selects the addressing mode on the Read Port of a channel. 设置的原理见 trm 11.4.3
//   0x0: Constant address mode        0x1: Post-incremented address mode
//   0x2: Single index address mode    0x3: Double index address mode
//这里设置 source 的参数主要是地址自动操作的模式及 element index 和 frame index
void omap_set_dma_src_params(int lch, int src_port, int src_amode,
                unsigned long src_start,
                int src_ei, int src_fi)
```

```
{
    u32 l;
...
    //设置 amode,自动地址的操作模式
    l = p -> dma_read( CCR, lch);
    l &= ~(0x03 << 12);
    l |= src_amode << 12;
    p -> dma_write(l, CCR, lch);

    //设置 source 的物理地址
    p -> dma_write( src_start, CSSA, lch);

    //设置 element index 和 frame index
    p -> dma_write( src_ei, CSEI, lch);
    p -> dma_write( src_fi, CSFI, lch);
}
EXPORT_SYMBOL( omap_set_dma_src_params);

//设置 DMA 的参数,对相应的驱动一次配置好所有 channel 的参数后使用
void omap_set_dma_params( int lch, struct omap_dma_channel_params * params)
{
    //设置传输参数
    omap_set_dma_transfer_params( lch, params -> data_type,
                                params -> elem_count, params -> frame_count,
                                params -> sync_mode, params -> trigger,
                                params -> src_or_dst_synch);
    //设置 source 的参数
    omap_set_dma _src_params( lch, params -> src_port,
                            params -> src_amode, params -> src_start,
                            params -> src_ei, params -> src_fi);
    //设置 target 的参数
    omap_set_dma_dest_params( lch, params -> dst_port,
                        params -> dst_amode, params -> dst_start,
                        params -> dst_ei, params -> dst_fi);
    //设置 read/write 优先级
    if( params -> read_prio || params -> write_prio)
        omap_dma_set_prio_lch( lch, params -> read_prio,params -> write_prio);
}
EXPORT_SYMBOL( omap_set_dma_params);

//设置是否 enable DMA 的 source 为 package 操作方式;package 方式可以将
//element 组成 package 以满足 DMA read/write port 的 size. 不支持 constant 地址方式
```

```
void omap_set_dma_src_data_pack(int lch, int enable)
{
    u32 l;

    l = p -> dma_read(CSDP, lch);
    l &= ~(1 << 6);
    //如果 enable 设置 source 为 pack enable
    if(enable)
        l |= (1 << 6);
    p -> dma_write(l, CSDP, lch);
}
EXPORT_SYMBOL(omap_set_dma_src_data_pack);

//dest_amode 的设置在 ccr 寄存器中
//Selects the addressing mode on the Read Port of a channel. 设置的原理见 trm 11.4.3
//  0x0: Constant address mode        0x1: Post - incremented address mode
//  0x2: Single index address mode    0x3: Double index address mode
//这里设置 destination 的参数主要是地址自动操作的模式及 element index 和 frame index
void omap_set_dma_dest_params(int lch, int dest_port, int dest_amode,
                              unsigned long dest_start,
                              int dst_ei, int dst_fi)
{
    u32 l;
...
    //设置地址操作模式
    l = p -> dma_read(CCR, lch);
    l &= ~(0x03 << 14);
    l |= dest_amode << 14;
    p -> dma_write(l, CCR, lch);

    //设置 destination start 物理地址
    p -> dma_write(dest_start, CDSA, lch);

    //设置 element index 和 frame index
    p -> dma_write(dst_ei, CDEI, lch);
    p -> dma_write(dst_fi, CDFI, lch);
}
EXPORT_SYMBOL(omap_set_dma_dest_params);

//设置是否 enable DMA 的 destination 为 package 操作方式;package 方式可以将
//element 组成 package 以满足 DMA read/write port 的 size. 不支持 constant 地址方式
void omap_set_dma_dest_data_pack(int lch, int enable)
```

```
{
      u32 l;

      l = p -> dma_read( CSDP, lch) ;
      l &= ~(1 << 13) ;
      //对 pack 模式进行设置
      if( enable)
            l |= 1 << 13;
      p -> dma_write(l, CSDP, lch) ;
}
EXPORT_SYMBOL( omap_set_dma_dest_data_pack) ;

//enable 相应的 channel 的 irq 中断,该操作会在 start_dma 时进行调用
static inline void omap_enable_channel_irq( int lch)
{
      u32 status;

      /* Clear CSR */
      //首先清除 channel status register
      if( cpu_class_is_omap1( ) )
            status = p -> dma_read( CSR, lch) ;
      else if( cpu_class_is_omap2( ) )
            p -> dma_write( OMAP2_DMA_CSR_CLEAR_MASK, CSR, lch) ;

      /* Enable some nice interrupts. */
      //enable 相应的 channle 中断,channel 的中断记录在 channel 的
      //管理逻辑结构中 dma_chan[ lch]. enabled_irqs
      p -> dma_write( dma_chan[ lch]. enabled_irqs, CICR, lch) ;
}

//disable 相应的 channel 中断
static void omap_disable_channel_irq( int lch)
{
      //写 0 为 disable 所有 channel 的中断
      if( cpu_class_is_omap2( ) )
            p -> dma_write(0, CICR, lch) ;
}

//设置 channel 想要 enable 的中断,这里只是设置 channel 中 eanble_irq 的属性,
//具体的 eanble 相应的中断会根据该属性进行 cicr 的配置
void omap_enable_dma_irq( int lch, u16 bits)
{
```

```
        dma_chan[lch].enabled_irqs |= bits;
    }
    EXPORT_SYMBOL(omap_enable_dma_irq);

    //设置 channel 想要 disable 的中断,这里只是设置 channel 中 eanble_irq 的属性,具体的 disable 相
    //应的中断会根据该属性进行 cicr 的配置
    void omap_disable_dma_irq(int lch, u16 bits)
    {
        dma_chan[lch].enabled_irqs &= ~bits;
    }
    EXPORT_SYMBOL(omap_disable_dma_irq);
```

3. 逻辑 channel 的操作接口

逻辑 channel 的操作接口主要是启动、停止以及中断响应接口。详细的解析如下:

```
    //omap 上 start DMA 的操作接口
    void omap_start_dma(int lch)
    {
        u32 l;

        /*
         * The CPC/CDAC register needs to be initialized to zero
         * before starting dma transfer.
         */
        //DMA start 之前 destination 地址计数寄存器需要清 0
        if(cpu_is_omap15xx())
            p->dma_write(0, CPC, lch);
        else
            p->dma_write(0, CDAC, lch);

        if(!omap_dma_in_1510_mode() && dma_chan[lch].next_lch != -1){
            //这里说明有 link 的 channel,需要设置 link 将
            //channel 串起来
            int next_lch, cur_lch;
            char dma_chan_link_map[dma_lch_count];

            //对 chain 使用的 channel 进行标记
            dma_chan_link_map[lch] = 1;
            /* Set the link register of the first channel */
            //设置当前 channel 的 next link,具体的 next link 在 dma_chan[lch].next_lch 中
            enable_lnk(lch);

            //重置 channel 标记
```

```
        memset(dma_chan_link_map, 0, sizeof(dma_chan_link_map));
        //下面是遍历 link,对 link 中每个 channel 进行设置
        cur_lch = dma_chan[lch].next_lch;
        do {
            next_lch = dma_chan[cur_lch].next_lch;

            /* The loop case: we've been here already */
            //如果 channel 已经设置,则说明是循环 link,退出
            if(dma_chan_link_map[cur_lch])
                break;
            /* Mark the current channel */
            //进行标记
            dma_chan_link_map[cur_lch] = 1;

            //设置 link
            enable_lnk(cur_lch);
            //enable channel 中的中断
            omap_enable_channel_irq(cur_lch);

            //遍历下一个
            cur_lch = next_lch;
        } while(next_lch != -1);
    } else if(IS_DMA_ERRATA(DMA_ERRATA_PARALLEL_CHANNELS))
        p->dma_write(lch, CLNK_CTRL, lch);

    //使能 channel 相应的 irq
    omap_enable_channel_irq(lch);

    //设置 channel ccr 寄存器来 enable 该 channel
    //对 software trigger 的 DMA channel,设置 ccr 寄存器的 enable bit
    //就直接触发 DMA 操作。如果是 hardware trigger 的 channel 会等待相应的
    //dma_req 信号触发
    l = p->dma_read(CCR, lch);

    if(IS_DMA_ERRATA(DMA_ERRATA_IFRAME_BUFFERING))
            l |= OMAP_DMA_CCR_BUFFERING_DISABLE;
    l |= OMAP_DMA_CCR_EN;

    //写相应的 enable bit
    p->dma_write(l, CCR, lch);

    //标记该 channel 为 active 状态
```

```
    dma_chan[lch].flags |= OMAP_DMA_ACTIVE;
}

//stop 相应 DMA channel 的接口函数
void omap_stop_dma(int lch)
{
    u32 l;
...
    l = p->dma_read(CCR, lch);
    if(IS_DMA_ERRATA(DMA_ERRATA_i541)&&
            (l & OMAP_DMA_CCR_SEL_SRC_DST_SYNC)){
        //这是对 source 同步 stop channel 时 bug 的 workaround
        int i = 0;
        u32 sys_cf;

        /* Configure No-Standby */
        l = p->dma_read(OCP_SYSCONFIG, lch);
        sys_cf = l;
        l &= ~DMA_SYSCONFIG_MIDLEMODE_MASK;
        l |= DMA_SYSCONFIG_MIDLEMODE(DMA_IDLEMODE_NO_IDLE);
        p->dma_write(l , OCP_SYSCONFIG, 0);

        l = p->dma_read(CCR, lch);
        l &= ~OMAP_DMA_CCR_EN;
        p->dma_write(l, CCR, lch);

        /* Wait for sDMA FIFO drain */
        l = p->dma_read(CCR, lch);
        while(i < 100 &&(l &(OMAP_DMA_CCR_RD_ACTIVE | OMAP_DMA_CCR_WR_AC-
TIVE))){
            udelay(5);
            i++;
            l = p->dma_read(CCR, lch);
        }
        if(i >= 100)
            printk(KERN_ERR "DMA drain did not complete on "
                        "lch %d\n", lch);
        /* Restore OCP_SYSCONFIG */
        p->dma_write(sys_cf, OCP_SYSCONFIG, lch);
    } else {
        //直接清除 ccr 寄存器的 enable bit 来 disable channel
        //从而 stop 相应的 channel
```

```
            l &= ~OMAP_DMA_CCR_EN;
            p->dma_write(l, CCR, lch);
        }

        if(!omap_dma_in_1510_mode()&& dma_chan[lch].next_lch != -1){
            //如果是 channel link,则要 disable link 中所有的 channel
            //主要是拆 link
            int next_lch, cur_lch = lch;
            char dma_chan_link_map[dma_lch_count];

            //清除相应的标记,以便检查所有的 link 都可以清除
            memset(dma_chan_link_map, 0, sizeof(dma_chan_link_map));
            do {
                /* The loop case: we've been here already */
                if(dma_chan_link_map[cur_lch])
                    break;
                /* Mark the current channel */
                //标记相应的 channel 已经 disable
                dma_chan_link_map[cur_lch] = 1;

                //disable 相应的 link,其中会置 channel 的 flag 为 deactive
                disable_lnk(cur_lch);

                //遍历 link
                next_lch = dma_chan[cur_lch].next_lch;
                cur_lch = next_lch;
            } while(next_lch != -1);
        }

        //设置 channel flag 为 deactive
        dma_chan[lch].flags &= ~OMAP_DMA_ACTIVE;
    }

    //下面开始是每个 DMA channel 的中断处理相关的函数
    static int omap2_dma_handle_ch(int ch)
    {
        //首先读取 channel 的中断状态寄存器
        u32 status = p->dma_read(CSR, ch);

        //如果是 csr 为 0 说明是 spurious DMA irq,报伪中断的错误
        if(!status){
            if(printk_ratelimit())
```

```
            printk(KERN_WARNING "Spurious DMA IRQ for lch %d\n",
                ch);
        //清除 DMA 对 MPU 相应的 channel 的状态位,以便后续中断处理
        p->dma_write(1 << ch, IRQSTATUS_L0, ch);
        return 0;
    }
    //channel 没有分配则报错
    if(unlikely(dma_chan[ch].dev_id == -1)){
        if(printk_ratelimit())
            printk(KERN_WARNING "IRQ %04x for non-allocated DMA"
                    "channel %d\n", status, ch);
        return 0;
    }
    //drop dma 处理则报错,出现该情况时需要加速处理
    //drop 硬件逻辑为 dma_req 来了但是之前还有 dma_req 没有处理
    if(unlikely(status & OMAP_DMA_DROP_IRQ))
        printk(KERN_INFO
                "DMA synchronization event drop occurred with device "
                "%d\n", dma_chan[ch].dev_id);
    //如果传输错误则报错
    if(unlikely(status & OMAP2_DMA_TRANS_ERR_IRQ)){
        printk(KERN_INFO "DMA transaction error with device %d\n",
                dma_chan[ch].dev_id);
        if(IS_DMA_ERRATA(DMA_ERRATA_i378)){
            //勘误中该中断发生硬件没有 disable channel
            //需要手动 disable
            u32 ccr;

            ccr = p->dma_read(CCR, ch);
            ccr &= ~OMAP_DMA_CCR_EN;
            p->dma_write(ccr, CCR, ch);
            dma_chan[ch].flags &= ~OMAP_DMA_ACTIVE;
        }
    }
    //secure 和 misaligned error 中断报错
    if(unlikely(status & OMAP2_DMA_SECURE_ERR_IRQ))
        printk(KERN_INFO "DMA secure error with device %d\n",
                dma_chan[ch].dev_id);
    if(unlikely(status & OMAP2_DMA_MISALIGNED_ERR_IRQ))
        printk(KERN_INFO "DMA misaligned error with device %d\n",
                dma_chan[ch].dev_id);
```

```
        //清除 channel status 状态
        p->dma_write(OMAP2_DMA_CSR_CLEAR_MASK, CSR, ch);
        //清除对 mpu 的 irq 状态
        p->dma_write(1 << ch, IRQSTATUS_L0, ch);
        /* read back the register to flush the write */
        //做读操作实际为 barrier 保证完成
        p->dma_read(IRQSTATUS_L0, ch);

        /* If the ch is not chained then chain_id will be -1 */
        if(dma_chan[ch].chain_id != -1){
            //如果是 chain,需要对 chain 进行处理
            //DM 3730 没有使用 chain,所以不会到这里
            int chain_id = dma_chan[ch].chain_id;
                dma_chan[ch].state = DMA_CH_NOTSTARTED;
            if(p->dma_read(CLNK_CTRL, ch)&(1 << 15))
                dma_chan[dma_chan[ch].next_linked_ch].state = DMA_CH_STARTED;
            if(dma_linked_lch[chain_id].chain_mode == OMAP_DMA_DYNAMIC_CHAIN)
                disable_lnk(ch);

            if(!OMAP_DMA_CHAIN_QEMPTY(chain_id))
                OMAP_DMA_CHAIN_INCQHEAD(chain_id);

            status = p->dma_read(CSR, ch);
        }

        //重写 status 则清除相应的状态 bit
        p->dma_write(status, CSR, ch);

        //调用 request 或者 set_dma_callback 时设置的 callback 函数
        //完成驱动的策略
        if(likely(dma_chan[ch].callback != NULL))
            dma_chan[ch].callback(ch, status, dma_chan[ch].data);

        return 0;
    }

    /* STATUS register count is from 1-32 while our is 0-31 */
    //MPU 处理 DMA channel 中断的接口函数
    static irqreturn_t omap2_dma_irq_handler(int irq, void *dev_id)
    {
        u32 val, enable_reg;
        int i;
```

```
    //首先读取 DMA irq status 寄存器
    val = p -> dma_read( IRQSTATUS_L0, 0) ;
    //为 0 标识欺骗中断报错
    if( val == 0) {
        if( printk_ratelimit( ) )
            printk( KERN_WARNING "Spurious DMA IRQ\n" ) ;
        return IRQ_HANDLED;
    }
    //只对已经 enable 的中断进行处理
    //要读取 irq enable 寄存器
    enable_reg = p -> dma_read( IRQENABLE_L0, 0) ;
    //对 status 进行处理保留 enable 的中断
    val & = enable_reg;/ *  Dispatch only relevant interrupts  * /
    for( i = 0; i < dma_lch_count && val ! = 0; i ++ ) {
        //移位处理每个置位的 channel
        //omap2_dma_handle_ch 为 channel 的中断处理函数
        if( val & 1)
            omap2_dma_handle_ch( i) ;
        val >>= 1;
    }

    return IRQ_HANDLED;
}

//DMA 于 kernel 的中断处理的接口,irqaction
static struct irqaction omap24xx_dma_irq = {
    . name = "DMA",
    . handler = omap2_dma_irq_handler,
    . flags = IRQF_DISABLED
};
```

4. SDMA 管理初始化及防止多核冲突的实现

SDMA 初始化还是以 platform 驱动的形式存在，而相应的防止多核使用 DMA 冲突的接口是 omap_dma_cmdline_reserve_ch，原理是通过启动参数设定 ARM MPU 保留的 DMA 逻辑 channel，使得 ARM MPU 只能使用保留的逻辑 channel。

```
//DMA 的 platform_driver 和 platform_device 进行绑定的接口,主要负责通过
//platform_device 的信息初始化相应的管理实体
static int __ devinit omap_system_dma_probe( struct platform_device  * pdev)
{
    int ch, ret = 0;
```

```
int dma_irq;
char irq_name[4];
int irq_rel;

//获得 DMA platform_device 的信息,主要是 resource 以及 channel 数还有 chip 的操作接口等信息
p = pdev -> dev. platform_data;
if(!p){
        dev_err(&pdev -> dev, "%s: System DMA initialized without"
                "platform data\n", __func__);
        return -EINVAL;
}

//包括 chip 级别的操作接口等信息
d               = p -> dma_attr;
errata          = p -> errata;

//检查是否 reserve channel,这些 channel 给 ARM 使用
if((d -> dev_caps & RESERVE_CHANNEL) && omap_dma_reserve_channels
            &&(omap_dma_reserve_channels <= dma_lch_count))
        d -> lch_count = omap_dma_reserve_channels;

dma_lch_count           = d -> lch_count;
dma_chan_count          = dma_lch_count;
//channel 的管理实体在 chip 的 platform_device 的初始化时候已经分配
//这里是个指针
dma_chan                = d -> chan;
enable_1510_mode = d -> dev_caps & ENABLE_1510_MODE;

if(cpu_class_is_omap2()){
        //分配 chain 中的 link 管理实体,DM 3730 上并没有使用 chain
        dma_linked_lch = kzalloc(sizeof(struct dma_link_info) *
                        dma_lch_count, GFP_KERNEL);
        if(!dma_linked_lch){
                ret = -ENOMEM;
                goto exit_dma_lch_fail;
        }
}

//下面对每个 channel 的管理实体进行初始化
spin_lock_init(&dma_chan_lock);
for(ch = 0; ch < dma_chan_count; ch++){
        //首先 clear 相应的每个 channle 寄存器
```

```
            omap_clear_dma(ch);
            if(cpu_class_is_omap2())
                omap2_disable_irq_lch(ch);

            //进行未分配和未建 link 的初始化
            dma_chan[ch].dev_id = -1;
            dma_chan[ch].next_lch = -1;

            if(ch >=6 && enable_1510_mode)
            continue;
        }

        //设置和性能相关的 gcr 参数,这里的设置和性能有关系
        if(cpu_is_omap2430() || cpu_is_omap34xx() || cpu_is_omap44xx())
            omap_dma_set_global_params(DMA_DEFAULT_ARB_RATE, DMA_DEFAULT_FIFO_
DEPTH, 0);

        //对中断进行初始化
        if(cpu_class_is_omap2()){
            //首先获得中断号
            strcpy(irq_name, "0");
            dma_irq = platform_get_irq_byname(pdev, irq_name);
            if(dma_irq <0){
                dev_err(&pdev->dev, "failed: request IRQ %d", dma_irq);
                goto exit_dma_lch_fail;
            }
            //设置中断处理接口
            ret = setup_irq(dma_irq, &omap24xx_dma_irq);
            if(ret){
                dev_err(&pdev->dev, "set_up failed for IRQ %d"
                    "for DMA(error %d)\n", dma_irq, ret);
                goto exit_dma_lch_fail;
            }
        }

        /* reserve dma channels 0 and 1 in high security devices */
        if(cpu_is_omap34xx()&&
            (omap_type()!=OMAP2_DEVICE_TYPE_GP)){
            //omap3hs 设备要保留 channel 0 和 1 给 HS ROM code
            printk(KERN_INFO "Reserving DMA channels 0 and 1 for "
                    "HS ROM code\n");
            dma_chan[0].dev_id =0;
```

```
        dma_chan[1].dev_id = 1;
    }
    //chip 的 DMA 能力显示
    p->show_dma_caps();
    return 0;

exit_dma_irq_fail:
    dev_err(&pdev->dev, "unable to request IRQ %d"
            "for DMA(error %d)\n", dma_irq, ret);
    for(irq_rel = 0; irq_rel < ch;irq_rel ++ ){
        dma_irq = platform_get_irq(pdev, irq_rel);
        free_irq(dma_irq,(void *)(irq_rel + 1));
    }

exit_dma_lch_fail:
    kfree(p);
    kfree(d);
    kfree(dma_chan);
    return ret;
}

//omap DMA 的 platform_driver 接口用于相关的管理实体的初始化和释放
static struct platform_driver omap_system_dma_driver = {
    .probe          = omap_system_dma_probe,
    .remove         = omap_system_dma_remove,
    .driver         = {
        .name       = "omap_dma_system"
    },
};

/*
 * Reserve the omap SDMA channels using cmdline bootarg
 * "omap_dma_reserve_ch =". The valid range is 1 to 32
 */
//为 bootargs 保留的 DMA channel reserve 的接口
static int __init omap_dma_cmdline_reserve_ch(char *str)
{
    if(get_option(&str, &omap_dma_reserve_channels) != 1)
        omap_dma_reserve_channels = 0;
    return 1;
}
__setup("omap_dma_reserve_ch =", omap_dma_cmdline_reserve_ch);
```

DMA 是很多驱动都需要使用的基础功能之一，以上进行了详细的代码级分析。以上的代码与 DMA engine 框架的功能类似，可以看做是 DMA engine 的芯片级实现。需要注意的是 DM 816X 系列芯片内部不是 SDMA 而是 EDMA，与 SDMA 差别主要是操作的寄存器不同，但是由于相应的设备驱动已经使用 omap 的 DMA 接口实现，最简单的移植就是将EDMA 的操作以 omap 的 DMA 接口封装，相应的 sdma2edma. c 就是实现该功能。

4. 6 时钟（clock）

4. 6. 1 clock 管理基本需求

时钟是硬件模块的基础，硬件模块要想正常的工作都需要系统为其提供时钟。当然系统很难为每个外设从单独的时钟源提供需要的时钟，而是通过如图 2-8 所示的时钟树的方式为每个硬件模块提供所需的时钟。

任何一个时钟应该都可以开关，设置频率。由于某个时钟是在时钟树上的一点，所以当改变某个时钟频率后会影响到该时钟下层的时钟频率，这就需要时钟管理以层次的方式进行，实现一种树状的管理。

实现了时钟的管理，就需要框架提供正确查找时钟的功能。总体来说，时钟需要能够进行组织管理，要能查找、能设置，设置要有继承性。另外作为一种资源，clock 实际是设备模块的一部分，所以时钟还要能够和设备产生关联，相应的也要有使用计数的管理。

4. 6. 2 clock 管理框架介绍

1. clock 控制结构及相关接口

Linux 内核中时钟的管理也是经历了时间的演进。早期的时钟管理，Linux 内核只是提供了接口的声明，并没有进行接口的定义和实现，相应的声明是在 linux/include/linux/clk. h 中，这些声明需要体系结构代码进行具体实现，这些声明为框架和具体的实现指明了方向。先来看看这些声明及其具体的意义（保留英文注释，因为解释已经足够详细）。

```
/*
 * struct clk - an machine class defined object/cookie.
 */
struct clk;

/**
 * clk_get - lookup and obtain a reference to a clock producer.
 * @dev: device for clock "consumer"
 * @id: clock comsumer ID
 *
 * Returns a struct clk corresponding to the clock producer, or
 * valid IS_ERR() condition containing errno.  The implementation
 * uses @dev and @id to determine the clock consumer, and thereby
```

```
 * the clock producer.  (IOW, @id may be identical strings, but
 * clk_get may return different clock producers depending on @dev.)
 *
 * Drivers must assume that the clock source is not enabled.
 *
 * clk_get should not be called from within interrupt context.
 */
struct clk *clk_get(struct device *dev, const char *id);

/**
 * clk_enable - inform the system when the clock source should be running.
 * @clk: clock source
 *
 * If the clock can not be enabled/disabled, this should return success.
 *
 * Returns success(0) or negative errno.
 */
int clk_enable(struct clk *clk);

/**
 * clk_disable - inform the system when the clock source is no longer required.
 * @clk: clock source
 *
 * Inform the system that a clock source is no longer required by
 * a driver and may be shut down.
 *
 * Implementation detail: if the clock source is shared between
 * multiple drivers, clk_enable() calls must be balanced by the
 * same number of clk_disable() calls for the clock source to be
 * disabled.
 */
void clk_disable(struct clk *clk);

/**
 * clk_get_rate - obtain the current clock rate(in Hz) for a clock source.
 *   This is only valid once the clock source has been enabled.
 * @clk: clock source
 */
unsigned long clk_get_rate(struct clk *clk);

/**
 * clk_put - "free" the clock source
```

```
 * @clk: clock source
 *
 * Note: drivers must ensure that all clk_enable calls made on this
 * clock source are balanced by clk_disable calls prior to calling
 * this function.
 *
 * clk_put should not be called from within interrupt context.
 */
void clk_put(struct clk *clk);

/*
 * The remaining APIs are optional for machine class support.
 */

/**
 * clk_round_rate - adjust a rate to the exact rate a clock can provide
 * @clk: clock source
 * @rate: desired clock rate in Hz
 *
 * Returns rounded clock rate in Hz, or negative errno.
 */
long clk_round_rate(struct clk *clk, unsigned long rate);

/**
 * clk_set_rate - set the clock rate for a clock source
 * @clk: clock source
 * @rate: desired clock rate in Hz
 *
 * Returns success(0) or negative errno.
 */
int clk_set_rate(struct clk *clk, unsigned long rate);

/**
 * clk_set_parent - set the parent clock source for this clock
 * @clk: clock source
 * @parent: parent clock source
 *
 * Returns success(0) or negative errno.
 */
int clk_set_parent(struct clk *clk, struct clk *parent);

/**
```

```
 * clk_get_parent - get the parent clock source for this clock
 * @clk: clock source
 *
 * Returns struct clk corresponding to parent clock source, or
 * valid IS_ERR() condition containing errno.
 */
struct clk *clk_get_parent(struct clk *clk);

/**
 * clk_get_sys - get a clock based upon the device name
 * @dev_id: device name
 * @con_id: connection ID
 *
 * Returns a struct clk corresponding to the clock producer, or
 * valid IS_ERR() condition containing errno.  The implementation
 * uses @dev_id and @con_id to determine the clock consumer, and
 * thereby the clock producer. In contrast to clk_get() this function
 * takes the device name instead of the device itself for identification.
 *
 * Drivers must assume that the clock source is not enabled.
 *
 * clk_get_sys should not be called from within interrupt context.
 */
struct clk *clk_get_sys(const char *dev_id, const char *con_id);

/**
 * clk_add_alias - add a new clock alias
 * @alias: name for clock alias
 * @alias_dev_name: device name
 * @id: platform specific clock name
 * @dev: device
 *
 * Allows using generic clock names for drivers by adding a new alias.
 * Assumes clkdev, see clkdev.h for more info.
 */
int clk_add_alias(const char *alias, const char *alias_dev_name, char *id,
            struct device *dev);
```

Linux 内核规定了这些接口的形式，特别是时钟的管理结构 struct clk 也只是进行了声明，它们都需要在体系结构进行具体的定义。从这些声明中可见，体系结构代码需要实现时钟的获得、频率设置、引用计数管理、管理层次的设置等功能。这些功能符合之前对需求的讨论。当然这些接口并不都需要实现。驱动设备的开发者只要调用上面的接口进行相关的操作即可，不必关心具体的实现。

Linux 内核这样的设计给各个体系结构很大的自主设计自由，主要是因为包括 Intel 在内的很多处理器，其相应的时钟树并不复杂。这样只需要进行统一的声明并由体系结构自己实现。

2. clock 查找和引用计数框架

上面的接口在 ARM 体系结构中实现了通用的时钟查找和引用计数的框架。具体的实现在 arch/arm/common/clkdev. c 中。针对时钟查找，ARM 体系结构定义了 struct clk_lookup 来实现。clk_lookup，顾名思义是用来查找 struct clk 结构的。有了它，就可以通过设备名或时钟的名字来找到相应的 struct clk 结构。clk_lookup 的组织框图如图 4-46 所示。

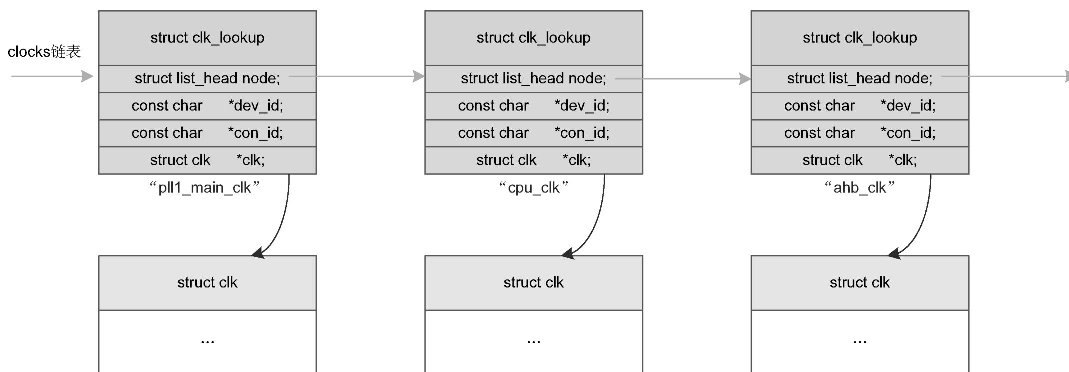


图 4-46　clk_lookup 的组织框图

从图 4-46 可见，clk_lookup 组织成链表，相应的每个节点都会指向一个 struct clk。这样就可以通过比对 clk_lookup 中的 dev_id 和 con_id 来找到正确的 struct clk 了。ARM 体系结构中通过 clkdev_add 将已经定义的 clk_lookup 加入链表进行管理，相应的需要芯片特定的代码对所有时钟对应的 clk_lookup 都进行定义。

Linux 内核中其他的时钟管理接口是在芯片特殊代码中实现的。

随着 SoC 的发展以及电源管理的需要，时钟树变得越来越复杂，内核也越来越需要统一时钟管理框架。新的 Linux 内核选择了芯片厂商 ST 提供的时钟框架作为通用时钟管理框架（common clock），其中对于时钟查找沿用 ARM 体系结构的 clk_lookup，并进行了些修改，struct clk 及其他接口都使用 ST 的实现。common clock 框架与 TI 芯片的时钟管理框架的实现整体上是一致的，所以这里不进行详细介绍，了解了 TI 芯片的时钟管理框架之后，再看 common clock 的框架就比较容易理解了。

4.6.3　TI 芯片 clock 管理相关实现详解

TI 芯片（包括 DM 3730 和 DM 816X）时钟管理框架的实现也是分层次的。在上层实现了 Linux 内核提供的接口函数作为抽象层，供其他模块使用；底层则是和芯片相关的具体接口实现。无论哪一层，相应的核心结构都是 struct clk，该结构在 arch/arm/plat-omap/include/plat/clock. h 中定义，具体内容如下：

```
struct clk {
    struct list_head      node;
    const struct clkops   * ops;
```

```
        const char              * name;
        struct clk        * parent;
        struct list_head      children;
        struct list_head      sibling;   /*  node for children  */
        unsigned long         rate;
        void __ iomem         * enable_reg;
        unsigned long         ( * recalc)(struct clk  * );
        int                ( * set_rate)(struct clk  * , unsigned long);
        long               ( * round_rate)(struct clk  * , unsigned long);
        void               ( * init)(struct clk  * );
        u8                 enable_bit;
        s8                 usecount;
        u8                 fixed_div;
        u8                 flags;
#ifdef CONFIG_ARCH_OMAP2PLUS
        void __ iomem        * clksel_reg;
        u32                clksel_mask;
        const struct clksel    * clksel;
        struct dpll_data       * dpll_data;
        const char             * clkdm_name;
        struct clockdomain    * clkdm;
#else
        u8                    rate_offset;
        u8                    src_offset;
#endif
#if defined(CONFIG_PM_DEBUG)&& defined(CONFIG_DEBUG_FS)
        struct dentry * dent;/*  For visible tree hierarchy  */
#endif
};
```

其中包括了和树状管理相关的属性 parent、children 和 sibling；使用计数的属性 usecount；操作接口；以及特定时钟类型相关的属性，如 clksel、dpll_data 等。这里包含了完整的信息。时钟管理的具体实现就是围绕着该数据结构展开的。

下面按照层次来介绍时钟管理。

1. 抽象层实现

相应的抽象实现是在 linux/arch/arm/plat-omap/clock. c 文件中，具体实现了 Linux 内核中规定的一些接口，代码分析如下：

```
//下面是 Linux 标准的 clk API 的实现,定义在 include/linux/clk. h 中
//这些 API 都做了必要的 lock 保护,可以在 SMP 和 irq 等所有的上下文调用
//这些 API 会调用 chip 具体的 clock 的操作
```

```
//相应 clock enable 的接口
int clk_enable(struct clk  * clk)
{
    unsigned long flags;
    int ret = 0;

    if(clk == NULL  || IS_ERR(clk))
        return  - EINVAL;

    spin_lock_irqsave(&clockfw_lock, flags);
    //调用芯片具体的操作接口
    if(arch_clock -> clk_enable)
        ret = arch_clock -> clk_enable(clk);
    spin_unlock_irqrestore(&clockfw_lock, flags);

    return ret;
}

//clock disable 的接口
void clk_disable(struct clk  * clk)
{
    unsigned long flags;

    if(clk == NULL  || IS_ERR(clk))
        return;

    spin_lock_irqsave(&clockfw_lock, flags);
    if(clk -> usecount == 0){
        pr_err("Trying disable clock %s with 0 usecount\n",
                clk -> name);
        WARN_ON(1);
        goto out;
    }
    //调用芯片具体的操作接口
    if(arch_clock -> clk_disable)
        arch_clock -> clk_disable(clk);

out:
    spin_unlock_irqrestore(&clockfw_lock, flags);
}

//获得 clock 的 rate
```

```
unsigned long clk_get_rate(struct clk *clk)
{
    unsigned long flags;
    unsigned long ret = 0;

    if(clk == NULL || IS_ERR(clk))
        return 0;

    spin_lock_irqsave(&clockfw_lock, flags);
    ret = clk->rate;
    spin_unlock_irqrestore(&clockfw_lock, flags);

    return ret;
}

/*
 * Optional clock functions defined in include/linux/clk.h
 */
//根据要设置的 clock 的 rate 以及 chip 的 special 得出 chip 能接受的最接近的 rate
//注意这里只是计算近似 rate,并不进行实际的设置,设置在 clk_set_rate 中进行
long clk_round_rate(struct clk *clk, unsigned long rate)
{
    unsigned long flags;
    long ret = 0;

    if(clk == NULL || IS_ERR(clk))
        return ret;

    spin_lock_irqsave(&clockfw_lock, flags);
    //调用芯片具体的操作接口
    if(arch_clock->clk_round_rate)
        ret = arch_clock->clk_round_rate(clk, rate);
    spin_unlock_irqrestore(&clockfw_lock, flags);

    return ret;
}

//设置 clock 相应的 rate
int clk_set_rate(struct clk *clk, unsigned long rate)
{
    unsigned long flags;
    int ret = -EINVAL;
```

```
    if(clk == NULL || IS_ERR(clk))
        return ret;

    spin_lock_irqsave(&clockfw_lock, flags);
    //调用芯片具体的操作接口
    if(arch_clock -> clk_set_rate)
        ret = arch_clock -> clk_set_rate(clk, rate);
    if(ret == 0){
        //设置 rate 成功,则该 clock 的 rate 要重新计算一下,保证正确
        if(clk -> recalc)
            clk -> rate = clk -> recalc(clk);
        //由于 clock rate 变化,相应的 clock 下面的子 clock 全要重新计算 rate
        propagate_rate(clk);
    }
    spin_unlock_irqrestore(&clockfw_lock, flags);

    return ret;
}

//对 clock 设置新的 parent
int clk_set_parent(struct clk *clk, struct clk *parent)
{
    unsigned long flags;
    int ret = -EINVAL;

    if(clk == NULL || IS_ERR(clk) || parent == NULL || IS_ERR(parent))
        return ret;

    spin_lock_irqsave(&clockfw_lock, flags);
    //只有 clock 的使用计数为 0 才可以设置,enable/disable 时会对计数进行操作
    if(clk -> usecount == 0){
        //chip 相关的 set_parent
        if(arch_clock -> clk_set_parent)
            ret = arch_clock -> clk_set_parent(clk, parent);
        if(ret == 0){
            //正确设置则需要重新计算 rate,包括子 clock 的 rate 计算
            if(clk -> recalc)
                clk -> rate = clk -> recalc(clk);
            propagate_rate(clk);
        }
    } else
```

```
            ret = -EBUSY;
        spin_unlock_irqrestore(&clockfw_lock, flags);

        return ret;
}

//获得 parent clock
struct clk *clk_get_parent(struct clk *clk)
{
        return clk->parent;
}
```

2. 芯片时钟接口具体实现

在抽象层看到很多接口都是调用芯片的具体实现来执行的，DM 3730 和 DM 816X 相关的具体实现是在 mach-omap2/clock.c 中，操作接口综合成 clk_functions。

```
struct clk_functions omap2_clk_functions = {
        .clk_enable             = omap2_clk_enable,
        .clk_disable            = omap2_clk_disable,
        .clk_round_rate         = omap2_clk_round_rate,
        .clk_set_rate           = omap2_clk_set_rate,
        .clk_set_parent         = omap2_clk_set_parent,
        .clk_disable_unused     = omap2_clk_disable_unused,
};
```

相应的接口实现详细分析如下：

```
//为 plat-omap clock 框架提供的 clock disable 函数
//实际的操作逻辑为先 disable clock,如果必要使 clock domain 可 idle,最后 parent disable
void omap2_clk_disable(struct clk *clk)
{
        //使用计数为 0 时,才真正进行 clock 的 disable 操作.
        if(clk->usecount == 0){
                WARN(1, "clock: %s: omap2_clk_disable() called, but usecount "
                        "already 0?", clk->name);
                return;
        }

        pr_debug("clock: %s: decrementing usecount\n", clk->name);

        clk->usecount--;

        if(clk->usecount > 0)
```

```
                return;

        pr_debug("clock: %s: disabling in hardware\n", clk->name);

        //计数为0,调用 clkops 的 disable 函数.
        clk->ops->disable(clk);

        //所属的 clock domain 也要 disable 操作,主要是保证相应的 clock domain 的 idle mode 可以使
        //相应的 clock domain(包括 power domain)处于 idle 状态,从而相应的 idle 功能也要等待 power
        //domain 状态转换并记录. 如果 hwsup 模式为 autoidle,可以通过增加或移除与 mpu daomin
        //的 dependency 关系使得 module active 或者 idle.
        if(clk->clkdm)
                omap2_clkdm_clk_disable(clk->clkdm, clk);

        //parent clock 的引用计数同样要减,都在 disable 中进行
        if(clk->parent)
                omap2_clk_disable(clk->parent);
}

//主要是为 plat-omap 提供的 clock eanble 函数,也可以被其他模块调用。实际的操作逻辑为 parent 要
//enable,clock 所属的 clock domain 要可以 active,然后才能 enable 该 clock 并且保证 clock domain
//为 active 状态
int omap2_clk_enable(struct clk *clk)
{
        int ret;

        pr_debug("clock: %s: incrementing usecount\n", clk->name);

        clk->usecount++;

        if(clk->usecount > 1)
                return 0;

        //第一次使用需要真正的 enable 操作
        pr_debug("clock: %s: enabling in hardware\n", clk->name);

        //首先 parent clock 要进行 enable 操作.
        if(clk->parent){
                ret = omap2_clk_enable(clk->parent);
                if(ret){
                        WARN(1, "clock: %s: could not enable parent %s: %d\n",
                                clk->name, clk->parent->name, ret);
                        goto oce_err1;
```

```
        }
    }

    //所属的 clock domain 要 enable,主要是保证相应的 clock domain 的 idle mode 可以使相应的
    //clock domain(包括 power domain)处于 active 状态,从而激活功能,相应的也要等待 power
    //domain 状态转换并记录. 如果 hwsup 模式为 autoidle,可以通过增加或移除与 mpu daomin
    //的 dependency 关系使得 module active 或者 idle.
    if(clk->clkdm){
        ret = omap2_clkdm_clk_enable(clk->clkdm, clk);
        if(ret){
            WARN(1, "clock: %s: could not enable clockdomain %s: "
                 "%d\n", clk->name, clk->clkdm->name, ret);
            goto oce_err2;
        }
    }

    //此时 enable 该 clock,通常为 omap2_dflt_clk_enable,主要是对寄存器 enable 位置位
    ret = clk->ops->enable(clk);
    if(ret){
        WARN(1, "clock: %s: could not enable: %d\n", clk->name, ret);
        goto oce_err3;
    }

    return 0;

//中间过程如果失败,进行相应的回退操作.
oce_err3:
    if(clk->clkdm)
        omap2_clkdm_clk_disable(clk->clkdm, clk);
oce_err2:
    if(clk->parent)
        omap2_clk_disable(clk->parent);
oce_err1:
    clk->usecount--;

    return ret;
}

/* Given a clock and a rate apply a clock specific rounding function */
//找到 clock 能接受的接近并小于所期望 rate 的频率,返回得到的频率值,并不做设置
long omap2_clk_round_rate(struct clk *clk, unsigned long rate)
{
```

```
    if(clk->round_rate)
        return clk->round_rate(clk, rate);

    return clk->rate;
}

/* Set the clock rate for a clock source */
//设置指定的 rate
int omap2_clk_set_rate(struct clk *clk, unsigned long rate)
{
    int ret = -EINVAL;

    pr_debug("clock: set_rate for clock %s to rate %ld\n", clk->name, rate);

    /* dpll_ck, core_ck, virt_prcm_set; plus all clksel clocks */
    if(clk->set_rate)
        ret = clk->set_rate(clk, rate);

    return ret;
}

//当需要改变 clock 的 source 时钟的时候使用,设置新的 parent
int omap2_clk_set_parent(struct clk *clk, struct clk *new_parent)
{
    if(!clk->clksel)
        return -EINVAL;

    if(clk->parent == new_parent)
        return 0;

    return omap2_clksel_set_parent(clk, new_parent);
}

/*
 * OMAP2+ clock reset and init functions
 */

#ifdef CONFIG_OMAP_RESET_CLOCKS
//如果 omap chip 的 clock 没有被使用,则 disable 相应的 clock 的接口函数
void omap2_clk_disable_unused(struct clk *clk)
{
    u32 regval32, v;
```

```
        v = (clk -> flags & INVERT_ENABLE) ? (1 << clk -> enable_bit) : 0;

        regval32 = __raw_readl(clk -> enable_reg);
        if((regval32 &(1 << clk -> enable_bit)) == v)
            return;

        pr_debug("Disabling unused clock \"%s\"\n", clk -> name);
        if(cpu_is_omap34xx()) {
            omap2_clk_enable(clk);
            omap2_clk_disable(clk);
        } else {
            clk -> ops -> disable(clk);
        }
        if(clk -> clkdm != NULL)
            pwrdm_clkdm_state_switch(clk -> clkdm);
}
#endif
```

从代码中可见，对 clock 的操作中还要设置上层的 clock domain，这符合芯片的电源管理设计框架。

3. 不同时钟类型的具体实现

由之前的代码和 struct clk 中可见，操作接口有两种，一种是在 struct clkops 中表示的，另外一种是直接在 struct clk 中的，如 set_rate 等。这是为什么呢？仔细看 struct clkops 中的操作接口是和时钟 enable 和 disable 相关的，而 struct clk 中的操作接口是和频率设定相关的。把它们分开是因为这是两种完全不同的操作类型。对 enable 和 disable 相关的操作单独管理是由于很多模块会有不止一种时钟，通常会有接口时钟和功能时钟，而设备要能使用，需要在 enable 某种时钟时伴随着 enable 相应的伙伴时钟；或者某些时钟是由模块对外产生的，在 enable 时需要等待相应的模块到某种状态时才是真正的 enable，只有等待到相应的状态时相关的操作才完整。而对时钟频率的操作则相对直接，只要根据不同的时钟类型进行正确的操作即可。

DM 3730 提供了几类 enable 和 disable 的操作接口，具体如下：

```
const struct clkops clkops_omap2_dflt_wait = {
    .enable         = omap2_dflt_clk_enable,
    .disable    = omap2_dflt_clk_disable,
    .find_companion     = omap2_clk_dflt_find_companion,
    .find_idlest    = omap2_clk_dflt_find_idlest,
};

const struct clkops clkops_omap2_dflt = {
```

```
    .enable      = omap2_dflt_clk_enable,
    .disable     = omap2_dflt_clk_disable,
};
```

可见主要的接口是 enable 和 disable。下面来进行详细分析，在分析之前先进行说明，相应的接口主要操作的寄存器是功能时钟使能 FCLKEN、接口时钟使能 ICLKEN 和状态检查 IDLEST 等寄存器。

```
//默认的 clock enable 函数
int omap2_dflt_clk_enable(struct clk *clk)
{
    u32 v;

    if(unlikely(clk->enable_reg == NULL)){
        pr_err("clock.c: Enable for %s without enable code\n",
               clk->name);
        return 0;/* REVISIT: -EINVAL */
    }

    //读取 enable 寄存器的值,并根据 flag 标识来设置 enable 相应的 clock
    //INVERT_ENABLE 标识 0 为 enable
    v = __raw_readl(clk->enable_reg);
    if(clk->flags & INVERT_ENABLE)
        v &= ~(1 << clk->enable_bit);
    else
        v |= (1 << clk->enable_bit);
    __raw_writel(v, clk->enable_reg);

    //设置 barrier 来保证之后的操作该 clock 已经 enable
    v = __raw_readl(clk->enable_reg);/* OCP barrier */

    //如果有 IDLEST 相关的操作,说明要等待 IDLEST 状态标识 module 为 ready
    if(clk->ops->find_idlest)
        _omap2_module_wait_ready(clk);

    return 0;
}

//默认的 clock disable 函数
void omap2_dflt_clk_disable(struct clk *clk)
{
    u32 v;
```

```
    if(!clk->enable_reg) {
        /*
         *' Independent here refers to a clock which is not
         * controlled by its parent.
         */
        printk(KERN_ERR "clock: clk_disable called on independent "
                "clock %s which has no enable_reg\n", clk->name);
        return;
    }

    //读取 enable 寄存器并设置为 disable
    v = __raw_readl(clk->enable_reg);
    if(clk->flags & INVERT_ENABLE)
        v |= (1 << clk->enable_bit);
    else
        v &= ~(1 << clk->enable_bit);
    __raw_writel(v, clk->enable_reg);
    /* No OCP barrier needed here since it is a disable operation */
    //disable 操作不需要 barrier
}

//查找 companion clock 的地址和相应的 enable bit. companion clock 就是
//转换 CM_ICLKEN* <-> CM_FCLKEN*
//有些 module 是没有 companion clock,比如 mailbox 只有 interface clock 的模块
void omap2_clk_dflt_find_companion(struct clk *clk, void __iomem **other_reg,
                                   u8 *other_bit)
{
    u32 r;

    /*
     * Convert CM_ICLKEN* <-> CM_FCLKEN*.  This conversion assumes
     * it's just a matter of XORing the bits.
     */
    //转换 FCLKEN 和 ICLKEN,并返回寄存器地址和相应的位域
    r = ((__force u32)clk->enable_reg ^(CM_FCLKEN ^ CM_ICLKEN));

    *other_reg = (__force void __iomem *)r;
    *other_bit = clk->enable_bit;
}

//返回 IDLEST 相关的寄存器、位域,以及非 ready to access 状态时的值.
```

```
//这里假设 IDLEST 所在的寄存器和 FCLKEN 有对应关系,只做简单的偏移即可
//但有些 module 不是这样
void omap2_clk_dflt_find_idlest(struct clk *clk, void __iomem **idlest_reg,
                    u8 *idlest_bit, u8 *idlest_val)
{
    u32 r;

    r = (((__force u32)clk->enable_reg & ~0xf0) | 0x20);
    *idlest_reg = (__force void __iomem *)r;
    *idlest_bit = clk->enable_bit;

    /*
     * 24xx uses 0 to indicate not ready, and 1 to indicate ready.
     * 34xx reverses this, just to keep us on our toes
     * AM35xx uses both, depending on the module.
     */
    //返回指示 not ready 的值 omap3 该值为 1
    if(cpu_is_omap24xx())
        *idlest_val = OMAP24XX_CM_IDLEST_VAL;
    else if(cpu_is_omap34xx())
        *idlest_val = OMAP34XX_CM_IDLEST_VAL;
    else
        BUG();
}
```

针对时钟频率的操作，则要考虑两种类型的时钟，分别是通过寄存器选择的 clock 和 dpll clock，这两种类型的时钟对应频率的操作是完全不同的。

先看看通过寄存器选择的时钟相关的操作接口实现：

```
//所谓 clksel clock 就是符合下面条件的 clock
//① parent clock 不是固定的
//② 包含 divider 因子的 clock
//③ 两者皆是
//其中 parent 以及 mux divider 的设置都是在 struct clksel* data structures
//multiplexer 就是对 parent 的选择,相应的 divider 存在与 parent 关联的结构 clksel_rate 中
//由于 clock 以及其 rate 属性并不是所有 SOC 都相同,为了保证代码重用,通过 cpu_mask 描述究
//竟是哪个 SOC,相应的 rate 中有 flag 表明那个 SOC 支持该值

//clksel_reg 寄存器地址和 clksel_mask 都在 struct clksel 中

//实际的物理寄存器是 PRCM 中的 CM 模块的各个 module 的 clksel 寄存器,比如 MPU_CM、IVA_CM
//等,PRCM 中的 PRM 也有 clock 相关的寄存器 Clock_Control_Reg_PRM,其中也包含 clksel 寄存器
```

```
//设置 clksel clock 的 rate. 只有 struct clk 中有 clksel 和 clksel_mask 才是 clksel 类型的 clock
int omap2_clksel_set_rate(struct clk *clk, unsigned long rate)
{
    u32 field_val, validrate, new_div = 0;

    if(!clk->clksel || !clk->clksel_mask)
        return -EINVAL;

    //首先验证获得的 rate 是否符合
    validrate = omap2_clksel_round_rate_div(clk, rate, &new_div);
    if(validrate != rate)
        return -EINVAL;

    //获得相应的寄存器应该填写的值
    field_val = _divisor_to_clksel(clk, new_div);
    if(field_val == ~0)
        return -EINVAL;

    //写寄存器,并配置 struct clk 的 rate 值
    _write_clksel_reg(clk, field_val);

    clk->rate = clk->parent->rate/new_div;

    pr_debug("clock: %s: set rate to %ld\n", clk->name, clk->rate);

    return 0;
}

//当前 omap clock 代码假设 parent 的设置都是 clksel 类型 clock
//即只有 clksel clock 可以设置 parent
//为 clock 设置新的 parent
int omap2_clksel_set_parent(struct clk *clk, struct clk *new_parent)
{
    u32 field_val = 0;
    u32 parent_div;

    if(!clk->clksel || !clk->clksel_mask)
        return -EINVAL;

    //根据新的 parent 获得 divisor 值和寄存器应该填写的值
    parent_div = _get_div_and_fieldval(new_parent, clk, &field_val);
    if(!parent_div)
        return -EINVAL;
```

```
    //写寄存器并重建 parent 的链表
    _write_clksel_reg(clk, field_val);

    clk_reparent(clk, new_parent);

    //重新设置 clk 的频率
    /* CLKSEL clocks follow their parents' rates, divided by a divisor */
    clk->rate = new_parent->rate;

    if(parent_div > 0)
        clk->rate/ = parent_div;

    pr_debug("clock: %s: set parent to %s(new rate %ld)\n",
         clk->name, clk->parent->name, clk->rate);

    return 0;
}

//根据需要设定的 clock 频率找到相应的 divisor 值,返回 rounded clock rate 或者 -1 error
//divisor 值通过传址方式返回
u32 omap2_clksel_round_rate_div(struct clk *clk, unsigned long target_rate,
                u32 *new_div)
{
    unsigned long test_rate;
    const struct clksel *clks;
    const struct clksel_rate *clkr;
    u32 last_div = 0;

    if(!clk->clksel || !clk->clksel_mask)
        return ~0;

    pr_debug("clock: clksel_round_rate_div: %s target_rate %ld\n",
         clk->name, target_rate);

    *new_div = 1;

    //首先根据现在的 clock parent 获得相应的 clk_sel
    clks = _get_clksel_by_parent(clk, clk->parent);
    if(!clks)
        return ~0;

    //根据 clk_sel 的 divisor 值来计算 rate 是否能 round target rate
    for(clkr = clks->rates; clkr->div; clkr++){
        if(!(clkr->flags & cpu_mask))
```

```
            continue;

        /* Sanity check */
        if(clkr->div <= last_div)
            pr_err("clock: clksel_rate table not sorted "
                   "for clock %s", clk->name);

        last_div = clkr->div;

        //根据 parent 的 rate 和 divisor 值来计算相应的 clock 的 rate
        test_rate = clk->parent->rate/clkr->div;

        //如果 rate <= 目标 rate 即是找到了 round target rate
        if(test_rate <= target_rate)
            break;/* found it */
    }

    if(!clkr->div){
        pr_err("clock: Could not find divisor for target "
               "rate %ld for clock %s parent %s\n", target_rate,
               clk->name, clk->parent->name);
        return ~0;
    }

    //传址返回 divisor 值
    *new_div = clkr->div;

    pr_debug("clock: new_div = %d, new_rate = %ld\n", *new_div,
            (clk->parent->rate/clkr->div));

    //返回 clock 修改 divisor 值后的 rate 值
    return clk->parent->rate/clkr->div;
}

//根据 clock 和 target rate 查找 rounded rate,divider 数组必须从低到高排序
long omap2_clksel_round_rate(struct clk *clk, unsigned long target_rate)
{
    u32 new_div;

    return omap2_clksel_round_rate_div(clk, target_rate, &new_div);
}
```

DPLL 锁相环时钟会和芯片中究竟采用怎样的锁相环技术相关。DM 3730 中使用的 DPLL 结构如图 4-47 所示。图 4-47 引自《DM 3730 芯片手册》中第 299 页框图。

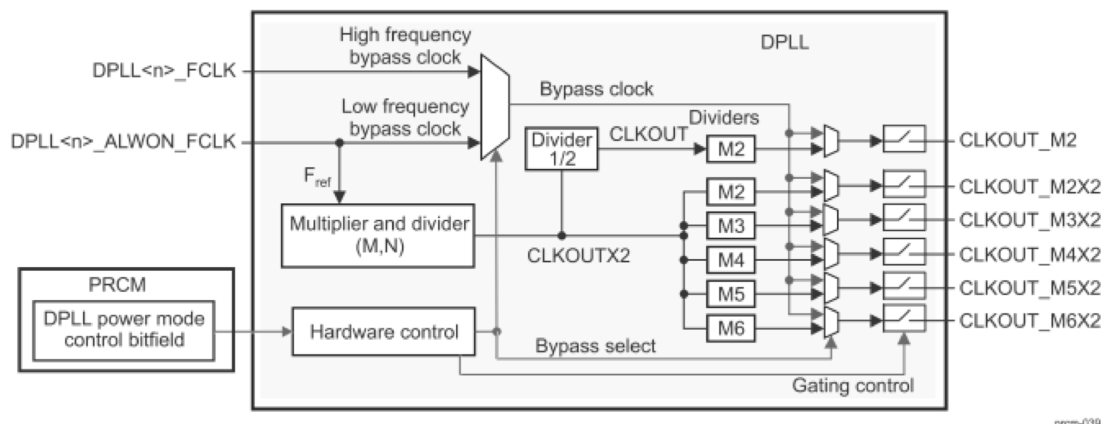


图 4-47　DM 3730 DPLL 结构图

由图 4-47 可见，DM 3730 的 DPLL 主要是设置乘数和除数（M 和 N），另外 DPLL 有 bypass 模式。操作接口分析如下：

```
//DPLL 的 set_rate 函数接口实现,根据频率可设置为 bypass 模式或者
//计算 M 和 N 的值后写 DPLL 并设置为 lock mode
//调用该函数后不应该影响相应的 bypass clock 和 ref clock 的引用计数和状态
int omap3_noncore_dpll_set_rate(struct clk *clk, unsigned long rate)
{
    struct clk *new_parent = NULL;
    u16 freqsel = 0;
    struct dpll_data *dd;
    int ret;

    //首先要确认是 DPLL 并且要设置 rate 与现在 DPLL rate 不同
    if(!clk || !rate)
        return -EINVAL;

    dd = clk->dpll_data;
    if(!dd)
        return -EINVAL;

    if(rate == omap2_get_dpll_rate(clk))
        return 0;

    /*
     * Ensure both the bypass and ref clocks are enabled prior to
     * doing anything; we need the bypass clock running to reprogram
```

```
 * the DPLL.
 */
//首先要 enable bypass clock 和 ref clock,由于重编 DPLL 时要使用 bypass clock
omap2_clk_enable(dd -> clk_bypass);
omap2_clk_enable(dd -> clk_ref);

//如果设置 rate 等于 bypass rate,则使用 bypass 模式
if(dd -> clk_bypass -> rate == rate &&
    (clk -> dpll_data -> modes &(1 << DPLL_LOW_POWER_BYPASS))){
    pr_debug("clock: %s: set rate: entering bypass. \n", clk -> name);

    ret = _omap3_noncore_dpll_bypass(clk);
    if(!ret)
        new_parent = dd -> clk_bypass;
} else {
//需要设置 M 和 N 为 lock 模式,首先要通过 omap2_dpll_round_rate 计算出 M 和 N 的值
    if(dd -> last_rounded_rate != rate)
        omap2_dpll_round_rate(clk, rate);

    //如果不能准确获得要设置的 rate 则返回错误
    if(dd -> last_rounded_rate == 0)
        return -EINVAL;

    /* No freqsel on OMAP4 and OMAP3630 */
    //只有 omap4 才需要计算并设置 freqsel
    if(!cpu_is_omap44xx()&& !cpu_is_omap3630()){
        freqsel = _omap3_dpll_compute_freqsel(clk, dd -> last_rounded_n);
        if(!freqsel)
            WARN_ON(1);
    }

    pr_debug("clock: %s: set rate: locking rate to %lu. \n",
         clk -> name, rate);

    //设置 M 和 N 并设置 lock 模式
    ret = omap3_noncore_dpll_program(clk, dd -> last_rounded_m,
                    dd -> last_rounded_n, freqsel);

    //记录新的 parent,后面会重新设置 parent
    if(!ret)
        new_parent = dd -> clk_ref;
}
```

```
    if( !ret) {
        //至此已经准确设置 rate
        /*
         * Switch the parent clock in the hierarchy, and make sure
         * that the new parent's usecount is correct.  Note: we
         * enable the new parent before disabling the old to avoid
         * any unnecessary hardware disable -> enable transitions.
         */
        //至此 clk 有了新的 parent(很大可能没有变化),需要将 old parent disable
        //为了避免同一个 parent clk 不必要的 disable -> enable 流程,在此先 enable
        if(clk -> usecount) {
            omap2_clk_enable(new_parent);
            omap2_clk_disable(clk -> parent);
        }

        //重新设置 parent 和 rate
        clk_reparent(clk, new_parent);
        clk -> rate = rate;
    }

    //调用该函数后不应该影响相应的 bypass clock 和 ref clock 的引用计数和状态,之前已经
    //enable,所以要 disable,保证调用此函数 clk 的状态没有变化.
    omap2_clk_disable(dd -> clk_ref);
    omap2_clk_disable(dd -> clk_bypass);

    return 0;
}

//计算 DPLL 的 rate 入口函数,负责计算出相应的 M 和 N,最后的 M 和 N 值会记录在 clk 中的 dpll_data 中
long omap2_dpll_round_rate(struct clk *clk, unsigned long target_rate)
{
    int m, n, r, e, scaled_max_m;
    unsigned long scaled_rt_rp, new_rate;
    int min_e = -1, min_e_m = -1, min_e_n = -1;
    struct dpll_data *dd;

    if( !clk || !clk -> dpll_data)
        return ~0;

    dd = clk -> dpll_data;

    pr_debug("clock: starting DPLL round_rate for clock %s, target rate "
```

```
        "%ld\n", clk->name, target_rate);

    //scaled_rt_rp 应该为 rate target 和 rate parent
    //首先通过因子把 multipy 放大,通过因式运算最终应该就是 target_rate
    scaled_rt_rp = target_rate/(dd->clk_ref->rate/DPLL_SCALE_FACTOR);
    scaled_max_m = dd->max_multiplier * DPLL_SCALE_FACTOR;

    dd->last_rounded_rate = 0;

    //开始测试合适的 M 和 N
    for(n = dd->min_divider; n <= dd->max_divider; n++){

        /* Is the(input clk, divider)pair valid for the DPLL? */
        //如果 N 导致内部 Fint 无效,则跳过;如果 N 已经导致内部 rate 太小,说明找不到合适的 N
        r = _dpll_test_fint(clk, n);
        if(r == DPLL_FINT_UNDERFLOW)
            break;
        else if(r == DPLL_FINT_INVALID)
            continue;

        /* Compute the scaled DPLL multiplier, based on the divider */
        //基于 N 来计算乘数,同样要放大.
        m = scaled_rt_rp * n;

        /*
         * Since we're counting n up, a m overflow means we
         * can bail out completely(since as n increases in
         * the next iteration, there's no way that m can
         * increase beyond the current m)
         */
        if(m > scaled_max_m)
            break;

        r = _dpll_test_mult(&m, n, &new_rate, target_rate,
                    dd->clk_ref->rate);

        /* m can't be set low enough for this n - try with a larger n */
        if(r == DPLL_MULT_UNDERFLOW)
            continue;

        //计算误差
        e = target_rate - new_rate;
```

```
            pr_debug("clock: n = %d: m = %d: rate error is %d "
                "(new_rate = %ld)\n", n, m, e, new_rate);

            if(min_e == -1 ||
                min_e >= (int)(abs(e) - dd -> rate_tolerance)) {
                min_e = e;
                min_e_m = m;
                min_e_n = n;

                //找到目前最好的 M 和 N 的值
                pr_debug("clock: found new least error %d\n", min_e);

                /* We found good settings -- bail out now */
                //如果可以接受,则跳出,最新的值记录在 min_e_m 和 min_e_n 中
                if(min_e <= dd -> rate_tolerance)
                    break;
            }
        }

        if(min_e < 0) {
            pr_debug("clock: error: target rate or tolerance too low\n");
            return ~0;
        }

        //找到相应的 M 和 N 记录在 dpll_data 中
        dd -> last_rounded_m = min_e_m;
        dd -> last_rounded_n = min_e_n;
        dd -> last_rounded_rate = _dpll_compute_new_rate(dd -> clk_ref -> rate,
                                            min_e_m,  min_e_n);

        pr_debug("clock: final least error: e = %d, m = %d, n = %d\n",
            min_e, min_e_m, min_e_n);
        pr_debug("clock: final rate: %ld  (target rate: %ld)\n",
            dd -> last_rounded_rate, target_rate);

        return dd -> last_rounded_rate;
    }
```

4. 时钟管理初始化

实际的芯片中所有时钟的信息都在 clockxxxx_data.c 内，DM 3730 的时钟信息是在 clock3xxx_data.c 内，其中不仅包含了详细的时钟信息，还包含芯片相关的时钟管理初始化操作。详细分析如下：

```
//该函数是 omap3 chip 的整个 clock tree 的初始化,包括 clock 数据结构的初始化和 clock tree 的
//建立,并且 enable 那些需要在 init 阶段使能的 clock。它是 omap 架构中重要的初始化函数
int __init omap3xxx_clk_init(void)
{
	struct omap_clk *c;
	u32 cpu_clkflg = 0;

	if(cpu_is_omap3517()){
		cpu_mask = RATE_IN_34XX;
		cpu_clkflg = CK_3517;
	} else if(cpu_is_omap3505()){
		cpu_mask = RATE_IN_34XX;
		cpu_clkflg = CK_3505;
	} else if(cpu_is_omap3630()){
		cpu_mask = (RATE_IN_34XX | RATE_IN_36XX);
		cpu_clkflg = CK_36XX;
	} else if(cpu_is_omap34xx()){
		if(omap_rev() == OMAP3430_REV_ES1_0){
			cpu_mask = RATE_IN_3430ES1;
			cpu_clkflg = CK_3430ES1;
		} else {
			/*
			 * Assume that anything that we haven't matched yet
			 * has 3430ES2-type clocks.
			 */
			cpu_mask = RATE_IN_3430ES2PLUS;
			cpu_clkflg = CK_3430ES2PLUS;
		}
	} else {
		WARN(1, "clock: could not identify OMAP3 variant\n");
	}

	if(omap3_has_192mhz_clk())
		omap_96m_alwon_fck = omap_96m_alwon_fck_3630;

	if(cpu_is_omap3630()){
		/*
		 * XXX This type of dynamic rewriting of the clock tree is
		 * deprecated and should be revised soon.
		 *
		 * For 3630: override clkops_omap2_dflt_wait for the
		 * clocks affected from PWRDN reset Limitation
```

```
		*/
	//对于 omap3630 需要调整相应的操作函数
	dpll3_m3x2_ck.ops =
			&clkops_omap36xx_pwrdn_with_hsdiv_wait_restore;
	dpll4_m2x2_ck.ops =
			&clkops_omap36xx_pwrdn_with_hsdiv_wait_restore;
	dpll4_m3x2_ck.ops =
			&clkops_omap36xx_pwrdn_with_hsdiv_wait_restore;
	dpll4_m4x2_ck.ops =
			&clkops_omap36xx_pwrdn_with_hsdiv_wait_restore;
	dpll4_m5x2_ck.ops =
			&clkops_omap36xx_pwrdn_with_hsdiv_wait_restore;
	dpll4_m6x2_ck.ops =
			&clkops_omap36xx_pwrdn_with_hsdiv_wait_restore;
}

/*
 * XXX This type of dynamic rewriting of the clock tree is
 * deprecated and should be revised soon.
 */
if(cpu_is_omap3630())
	dpll4_dd = dpll4_dd_3630;
else
	dpll4_dd = dpll4_dd_34xx;

//向 omap 系统 clock 驱动注册 chip 相关的通用上层操作接口
clk_init(&omap2_clk_functions);

//对每个 clock 进行 preinit
for(c = omap3xxx_clks; c < omap3xxx_clks + ARRAY_SIZE(omap3xxx_clks);
	 c++)
	clk_preinit(c->lk.clk);

//将每个 clock 加入系统管理注册,并将 clock 与所属的 clock domain 绑定
for(c = omap3xxx_clks; c < omap3xxx_clks + ARRAY_SIZE(omap3xxx_clks);
	 c++)
	if(c->cpu & cpu_clkflg){
		clkdev_add(&c->lk);
		clk_register(c->lk.clk);
		//初始化 clock 将 clock 与所属的 clock domain 绑定
		omap2_init_clk_clkdm(c->lk.clk);
	}
```

```
        //根据晶振的时钟遍历所有的时钟并计算频率
        recalculate_root_clocks();

        pr_info("Clocking rate(Crystal/Core/MPU): %ld.%01ld/%ld/%ld MHz\n",
            (osc_sys_ck.rate/1000000),(osc_sys_ck.rate/100000)% 10,
            (core_ck.rate/1000000),(arm_fck.rate/1000000));

        /*
         * Only enable those clocks we will need, let the drivers
         * enable other clocks as necessary
         */
        //对 clock 进行 enable 操作,如果 clock 标记为 kernel init 时 enable 则 enable 该 clock
        clk_enable_init_clocks();

        /*
         * Lock DPLL5 and put it in autoidle.
         */
        if(omap_rev() >= OMAP3430_REV_ES2_0)
            omap3_clk_lock_dpll5();

        /* Avoid sleeping during omap3_core_dpll_m2_set_rate() */
        sdrc_ick_p = clk_get(NULL, "sdrc_ick");
        arm_fck_p = clk_get(NULL, "arm_fck");

        return 0;
    }
```

注意，该操作是对整个系统的时钟的初始化函数，所以需要在系统初始化比较早的时候进行，对于 DM 3730 是在板级的中断初始化接口中调用。

至此，关于时钟管理的框架和主要接口分析完毕。

4.7　时间管理（Time）

4.7.1　时间管理基本需求

时间管理是操作系统实现的一个基本方面，在操作系统中对于时间管理的需求分为若干不同的类别：

- time keeping——计时。
- clock synchronization——时钟同步。
- time-of-day representation（TOD）——时间日期表示。
- next event interrupt scheduling——下一个事件中断调度。
- process and in-kernel timers——进程和内核定时器。
- process accounting——进程调度记录。

- process profiling——进程分析。

除了以上功能的需求还对时间精度有需求，需要能够支持各种时间精度的设备。

4.7.2 时间管理框架介绍

1. 老版本内核时间管理框架

在 Linux 2.6.16 之前，内核一直使用一种称为 timer wheel 的机制来管理时钟。这就是熟知的基于 Hz 的 timer 机制。timer wheel 有占用的内存少等优点，但 timer wheel 的实现机制仍然存在弊端。一方面 timer wheel 是为 timeout 类型的定时器优化的，并不适合精准定时 timer；另一方面，由于 timer wheel 是建立在 Hz 的基础上的，因此其计时精度无法进一步提高。毕竟一味的通过提高 Hz 值来提高计时精度并无意义，结果只能是产生大量的定时中断，增加额外的系统开销。因此，有必要将高精度的 timer 与低精度的 timer 分开，这样既可以确保低精度的 timeout 类型的定时器应用，也便于高精度的 timer 类型定时器的应用。另外 timer wheel 的实现与 jiffies 的耦合性太强，非常不便于扩展。

2. 新的时间管理框架

为了解决 timer wheel 低精度以及与内核其他模块的高耦合性的缺点，Linux 内核引入了 hrtimer。另外为了能在电源管理方面进行优化，有必要去除以频率（Hz）触发的定时时钟中断，改为动态时钟机制（dynamic tick）或者说 NO_HZ 机制。新的时间管理框架如图 4-48 所示。

下面介绍其中的各个模块。

① 时钟源设备（clock-source device）。它是系统中可以提供一定精度的计时设备。不同的时钟源提供的时钟精度是不一样的。此外，时钟源的计时都是单调递增的（monotonically）。时钟源作为系统时钟的提供者，在可靠并且可用的前提下精度越高越好。在 Linux 中不同的时钟源有不同的频率，具有更高频率的时钟源会优先被系统使用。

② 时钟事件设备（clock-event device）。它是系统中可以触发 one-shot（单次）或者周期性中断的设备。某些设备既可以做时钟源设备也可以做时钟事件设备。时钟事件设备的类型分为全局和 per-cpu 两种类型。全局的时钟事件设备完成的是系统相关的工作，例如完成系统的 tick 更新；per-cpu 的时钟事件设备主要完成本地 CPU 上的一些功能，例如对在当前 CPU 上运行进程的时间统计，profile，设置本地 CPU 上的下一次事件中断等。和时钟源设备的实现类似，时钟事件设备也通过频率来区分优先级关系。

③ tick device。它是用来处理周期性的 tick event。tick device 其实是对时钟事件设备的整合，因此 tick device 也有 one-shot 和周期性这两种中断触发模式。每注册一个时钟事件设备，这个设备会自动被注册为一个 tick device。全局的 tick device 用来更新诸如 jiffies 这样的全局信息，per-cpu 的 tick device 则用来更新每个 CPU 相关的特定信息。

④ hrtimer。它是建立在 per-cpu 时钟事件设备基础上的。对于一个 SMP 系统，如果只有全局的时钟事件设备，hrtimer 无法工作。ktime_t 是 hrtimer 主要使用的时间结构。无论使用哪种体系结构，ktime_t 始终保持 64bit 的精度，并且考虑了大小端的影响。hrtimer 有两种工作模式：低精度模式（low-resolution mode）与高精度模式（high-resolution mode）。虽然 hrtimer 子系统是为高精度的 timer 准备的，但是系统可能在运行过程中动态切换到不同精度的时钟源设备，因此 hrtimer 需要能够在低精度模式与高精度模式下自由切换。低精度模

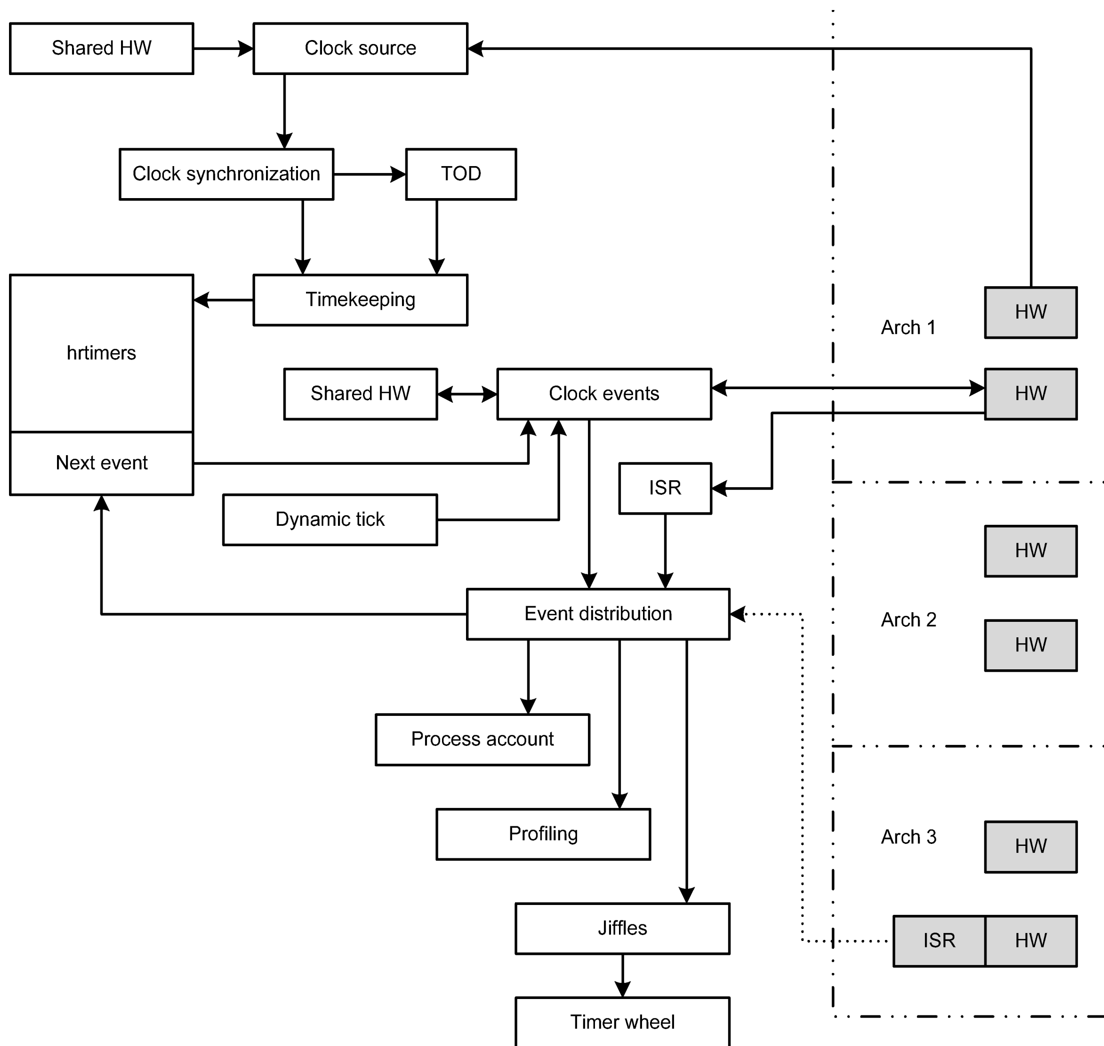


图 4-48　Linux 内核时间管理框架

式是建立在高精度模式基础之上的，在低精度模式下，hrtimer 的核心处理函数是 hrtimer_run_queues，每一次 tick 中断都要执行一次。hrtimer_bases 是实现 hrtimer 的核心数据结构，通过 hrtimer_bases，hrtimer 可以管理挂在每一个 CPU 上的所有 timer。在 update_process_times 中，除了处理处于低精度模式的 hrtimer 外，还要唤醒时间中断的 softirq（TIMER_SOFTIRQ）以便执行 timer wheel 的代码。由于 hrtimer 子系统的加入，在时间中断的 softirq 中还需要通过 hrtimer_run_pending 检查是否可以将 hrtimer 切换到高精度模式，如果可以将 hrtimer 切换到高精度模式，则调用 hrtimer_switch_to_hres 函数进行切换。

⑤ dynamic tick。它能在系统空闲时通过停止 tick 的运行以达到降低处理器功耗的目的。使用 dynamic tick 的系统，只有在实际工作时才会产生 tick，否则 tick 处于停止状态。dynamic tick 是为彻底替换掉周期性的 tick 机制而产生的。它对周期性运行的 tick 机制所完成的（如进程时间片的计算、更新 profile、协助 CPU 进行负载均衡等）诸多工作都提供了相应的模拟机制。hrtimer 从低精度模式切换到高精度模式的切换点，也是低精度模式下从

周期性 tick 到 dynamic tick 的切换点，会将 tick 切换到 one-shot 模式下，另外通过 tick_nohz_handler 模拟周期性 tick device 完成的工作：如果当前 CPU 负责全局 tick device 的工作，则更新 jiffies，同时完成对本地 CPU 的进程时间统计等工作；如果当前 tick device 在此之前已经处于停止状态，为了防止 tick 停止时间过长造成 watchdog 超时，从而引发 soft - lockdep 的错误，需要通过调用 touch_softlockup_watchdog 软件复位看门狗防止其溢出；设置了下一次的超时事件，但是由于系统空闲时会停止 tick，因此下一次的超时事件可能发生，也可能不发生。在高精度模式下 tick_sched_timer 用来模拟周期性 tick device 的功能。dynamic tick 的实现也使用了这个函数。这是因为 hrtimer 在高精度模式时必须使用 one - shot 模式的 tick device，这也同时符合 dynamic tick 的要求。虽然使用同样的函数，表面上都会触发周期性的 tick 中断，但是使用 dynamic tick 的系统在空闲时会停止 tick 工作，因此 tick 中断不会是周期产生的。对于 tick 的开关在 cpu idle 中通过 tick_nohz_stop_sched_tick 和 tick_nohz_restart_sched_tick 来实现，其中通过 tick_do_update_jiffies64 来更新时间，保证模拟的正确性。

4.7.3 TI 芯片时间管理相关实现详解

关于时间管理，Linux 内核已经提供了完整的解决方案，芯片本身的实现主要是 clock event 和 clock source 的硬件实现。下面基于 DM 3730 分析一下相关的实现。

1. clock event 相关实现

clock event 的相关实现如下：

```
//系统 clock event 的初始化接口
static void __init omap2_gp_clockevent_init(void)
{
    u32 tick_rate;
    int src;

    inited = 1;

    //申请特定的 gp timer
    gptimer = omap_dm_timer_request_specific(gptimer_id);
    BUG_ON(gptimer == NULL);
    //同样的 gp timer 作为 pm debug 的 wakeup 定时 timer
    gptimer_wakeup = gptimer;

    //指定 clock event 的 source clock
#if defined(CONFIG_OMAP_32K_TIMER)
    src = OMAP_TIMER_SRC_32_KHZ;
#else
    src = OMAP_TIMER_SRC_SYS_CLK;
    WARN(gptimer_id == 12, "WARNING: GPTIMER12 can only use the "
        "secure 32KiHz clock source\n");
#endif
```

```
    //设置 gp timer 的 source clock
    if(gptimer_id != 12)
        WARN(IS_ERR_VALUE(omap_dm_timer_set_source(gptimer, src)),
            "timer-gp: omap_dm_timer_set_source() failed\n");

    //获得 rate
    tick_rate = clk_get_rate(omap_dm_timer_get_fclk(gptimer));

    pr_info("OMAP clockevent source: GPTIMER%d at %u Hz\n",
        gptimer_id, tick_rate);

    //设置 irq 及 enable overflow 模式,真正的 gp timer start 是在 set_mode 的时候执行
    omap2_gp_timer_irq.dev_id = (void *)gptimer;
    setup_irq(omap_dm_timer_get_irq(gptimer), &omap2_gp_timer_irq);
    omap_dm_timer_set_int_enable(gptimer, OMAP_TIMER_INT_OVERFLOW);

    //设置 clock event 精度的属性,通常是 one_shot 的时钟设置 next event 时使用的,用来计算如
    //何配置时钟,对于周期性时钟没有进行这些计算
    clockevent_gpt.mult = div_sc(tick_rate, NSEC_PER_SEC,
                    clockevent_gpt.shift);
    clockevent_gpt.max_delta_ns =
        clockevent_delta2ns(0xffffffff, &clockevent_gpt);
    clockevent_gpt.min_delta_ns =
        clockevent_delta2ns(3, &clockevent_gpt);
        /* Timer internal resynch latency. */

    //注册相应的 clock event
    clockevent_gpt.cpumask = cpumask_of(0);
    clockevents_register_device(&clockevent_gpt);
}

//对于 kernel 的 clock event,主要是做 tick 的源.
//在 event_device add 的时候会通知 tick 管理层,然后设置周期方式,进而以一定周期发起时钟
//event。在时钟中断中通过 tick 层注册的 event_handler 来向 tick 层发布时钟时间,从而完成后
//续的 tick 及 timer 的处理
static struct clock_event_device clockevent_gpt = {
    .name       = "gp timer",
    .features       = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
    .shift      = 32,
    .set_next_event = omap2_gp_timer_set_next_event,
    .set_mode = omap2_gp_timer_set_mode,
```

```
};

//设置 timer 的 next event 的接口
static int omap2_gp_timer _set_next_event( unsigned long cycles,
                              struct clock_event_device  * evt)
{
    //设置 event 的时间并且 start 时钟
    omap_dm_timer_set_load_start( gptimer, 0, 0xffffffff - cycles);

    return 0;
}

//设置 mode 的接口,这里实际为 tick 源,所以主要是 CLOCK_EVT_MODE_PERIODIC
static void omap2_gp_timer_set_mode( enum clock_event_mode mode,
                                struct clock_event_device  * evt)
{
    u32 period;

    omap_dm_timer_stop( gptimer);

    switch( mode) {
    case CLOCK_EVT_MODE_PERIODIC:
        //设置周期值,即 1 s 的 tick 数换算的 timer 的加载值
        period = clk_get_rate( omap_dm_timer_get_fclk( gptimer))/HZ;
        period  -  = 1;
        //设置 timer 的计数值并且保证是 autoload 方式可以周期性触发中断
        omap_dm_timer_set_load_start( gptimer, 1, 0xffffffff - period);
        break;
    case CLOCK_EVT_MODE_ONESHOT:
        //不需要设置
        break;
    case CLOCK_EVT_MODE_UNUSED:
    case CLOCK_EVT_MODE_SHUTDOWN:
    case CLOCK_EVT_MODE_RESUME:
        break;
    }
}
```

2. clock source 相关实现

clock source 的相关实现如下：

```
//clock source 的初始化,主要是对属性的设置,并注册 clock source
static void __ init omap2_gp_clocksource_init( void)
```

```
{
    static struct omap_dm_timer *gpt;
    u32 tick_rate;
    static char err1[] __initdata = KERN_ERR
        "%s: failed to request dm-timer\n";
    static char err2[] __initdata = KERN_ERR
        "%s: can't register clocksource!\n";

    //获得 gp timer
    gpt = omap_dm_timer_request();
    if(!gpt)
        printk(err1, clocksource_gpt.name);
    gpt_clocksource = gpt;

    //设置 sys_clk 为其 source
    omap_dm_timer_set_source(gpt, OMAP_TIMER_SRC_SYS_CLK);
    //获得 sys_clk rate 以便后面计算相应属性
    tick_rate = clk_get_rate(omap_dm_timer_get_fclk(gpt));

    //从 0 计数,并 start gp timer
    omap_dm_timer_set_load_start(gpt, 1, 0);

    //计算 mult 属性
    clocksource_gpt.mult =
        clocksource_khz2mult(tick_rate/1000, clocksource_gpt.shift);
    //注册 clock source
    if(clocksource_register(&clocksource_gpt))
        printk(err2, clocksource_gpt.name);
}

//clock source 的属性
static struct clocksource clocksource_gpt = {
    .name        = "gp timer",
    .rating      = 300,
    .read   = clocksource_read_cycles,
    .mask        = CLOCKSOURCE_MASK(32),
    .shift       = 24,
    .flags       = CLOCK_SOURCE_IS_CONTINUOUS,
};

//查询 clock source 的计数,使用 gp timer 的时候是从 0 计数,所以可以很久才 overflow
static cycle_t clocksource_read_cycles(struct clocksource *cs)
```

```
{
        return(cycle_t)omap_dm_timer_read_counter(gpt_clocksource);
}
```

3. 初始化相关实现

整体的初始化是通过 system timer 来实现的，细节如下：

```
//Linux kernel 第一个注册的 clock_event_device 将作为 tick 源周期发布时钟中断
//在 omap3 上 gptimer1 是这个时钟,作为 machine_desc 的 sys_timer,初始化时会调用其 init
struct sys_timer omap_timer = {
        .init = omap2_gp_timer_init,
};

//作为 machine_desc 的 sys_timer 的初始化接口
static void __init omap2_gp_timer_init(void)
{
#ifdef CONFIG_LOCAL_TIMERS
        if(cpu_is_omap44xx()){
                twd_base = ioremap(OMAP44XX_LOCAL_TWD_BASE, SZ_256);
                BUG_ON(!twd_base);
        }
#endif
        //omap 平台的 dual mode timer 的初始化在 plat-omap/dmtimer 中
        omap_dm_timer_init();

        //clock event 和 clock source 初始化,如果是 32k 时钟作为 clock source 相关的初始化后续
        //作为 arch_initcall 执行.
        omap2_gp_clockevent_init();
        omap2_gp_clocksource_init();
}
```

关于 Linux 内核时间管理，处理器的代码只要实现以上部分就可以了，其余的部分都是由系统统一提供并实现的。

4.8 通用目的输入输出（GPIO）

GPIO 是 General Purpose Input/Output 的缩写，是一种灵活的可以通过软件进行控制的数字信号接口。现代的 SoC 都提供 GPIO，已经成为 SoC 一个强有力的扩展工具，可以通过 GPIO 以及软件实现一些接口功能模块，实现对外部器件进行控制。

4.8.1 GPIO 管理基本需求

GPIO 是通过芯片管脚来进行操作，可以作为输入也可以作为输出，作为输入的时候还可以作为中断源。

GPIO 也是一种公共资源，所以需要能够对 GPIO 进行统一的管理，能够申请和释放，还可以配置输入或者输出。针对 GPIO 作为中断源的情况，需要能够对该中断源进行管理，可以根据需要配置成边沿触发或者电平触发等不同的中断处理方式。

4.8.2　GPIO 管理框架介绍

关于 GPIO 管理，Linux 内核提供了 GPIO lib 来实现具体的 GPIO 服务，提供了统一的接口供其他模块使用，基本的框架如图 4-49 所示。

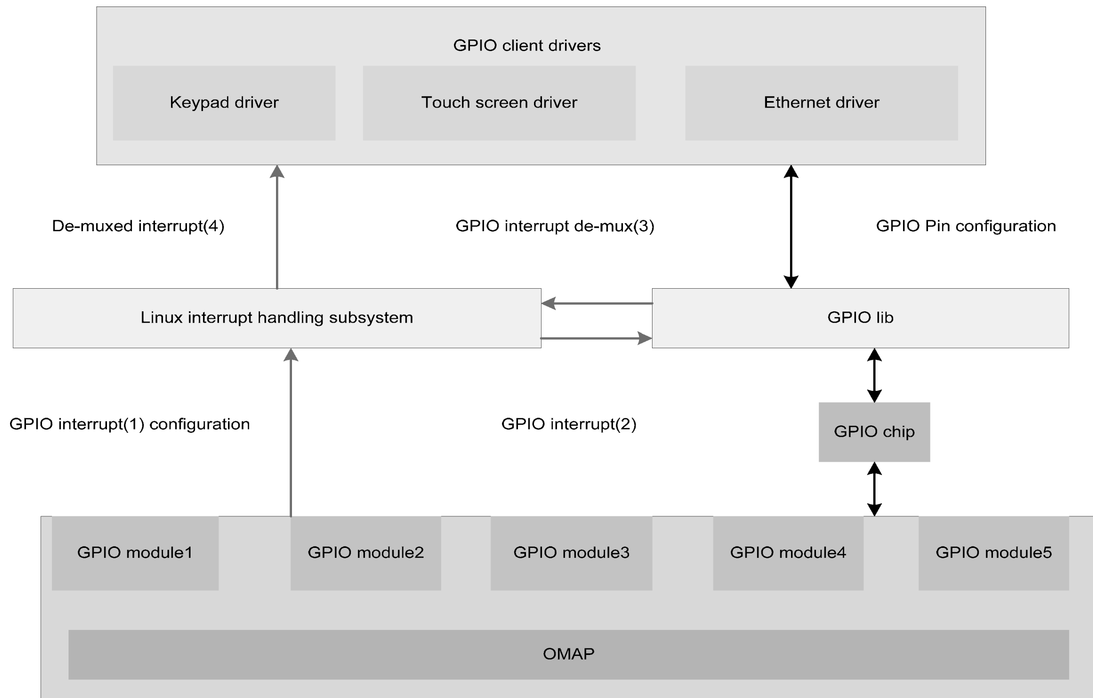


图 4-49　Linux 内核 GPIO 框架

从图 4-49 可见，各种设备模块都可以访问使用 GPIO，相应的需要通过 GPIO lib 和 Linux 内核中断处理子系统进行操作。而真正的 GPIO 芯片的实现是通过 GPIO chip 提供的接口来实现的。

1. 外部使用 GPIO 的操作接口

GPIO lib 提供的功能接口如下：

```
//对 GPIO 请求和释放的接口
extern int gpio_request(unsigned gpio, const char *label);
extern void gpio_free(unsigned gpio);
//设置 GPIO 引脚输入输出状态
extern int gpio_direction_input(unsigned gpio);
extern int gpio_direction_output(unsigned gpio, int value);
//由于 GPIO 信号可能出现抖动该接口设置去抖功能
extern int gpio_set_debounce(unsigned gpio, unsigned debounce);
```

```
//获得和设置 GPIO 引脚可以 sleep
extern int gpio_get_value_cansleep(unsigned gpio);
extern void gpio_set_value_cansleep(unsigned gpio, int value);

/* A platform's <asm/gpio.h> code may want to inline the I/O calls when
 * the GPIO is constant and refers to some always-present controller,
 * giving direct access to chip registers and tight bitbanging loops.
 */
//获得 GPIO 引脚的值
extern int __gpio_get_value(unsigned gpio);
//设置 GPIO 引脚的值
extern void __gpio_set_value(unsigned gpio, int value);
//检查 GPIO 是否可以 sleep
extern int __gpio_cansleep(unsigned gpio);
//GPIO 号转换成中断号
extern int __gpio_to_irq(unsigned gpio);
```

这些接口的名字都很好理解，相应的都会调用 GPIO chip 的接口函数。

2. GPIO 内部管理框架

GPIO 框架的重要数据结构就是 gpio_chip，详细内容如下：

```
struct gpio_chip {
    const char          *label;
    struct device       *dev;
    struct module       *owner;

    int         (*request)(struct gpio_chip *chip,
                        unsigned offset);
    void        (*free)(struct gpio_chip *chip,
                        unsigned offset);

    int         (*direction_input)(struct gpio_chip *chip,
                        unsigned offset);
    int         (*get)(struct gpio_chip *chip,
                        unsigned offset);
    int         (*direction_output)(struct gpio_chip *chip,
                        unsigned offset, int value);
    int         (*set_debounce)(struct gpio_chip *chip,
                        unsigned offset, unsigned debounce);

    void        (*set)(struct gpio_chip *chip,
                        unsigned offset, int value);
```

```
    int ( * to_irq)(struct gpio_chip  * chip,
                        unsigned offset);

    void ( * dbg_show)(struct seq_file  * s,
                        struct gpio_chip  * chip);
    //管理 GPIO 的起始号
    int base;
    //一共管理 GPIO 的数目
    u16 ngpio;
    const char        * const  * names;
    unsigned      can_sleep:1;
    unsigned      exported:1;
};
```

系统中可以有多个 gpio_chip，而每个 gpio_chip 管理一组 GPIO，其中 base 表示相应的 gpio_chip 管理的系统中的起始 GPIO 号，一共管理 ngpio 个 GPIO。

内核将所有的 GPIO 组成数组，每个 GPIO 用 gpio_desc 进行描述，具体的结构如下：

```
struct gpio_desc {
    struct gpio_chip * chip;
    unsigned longflags;
/ *  flag symbols are bit numbers  * /
#define FLAG_REQUESTED   0
#define FLAG_IS_OUT   1
#define FLAG_RESERVED   2
#define FLAG_EXPORT   3   / *  protected by sysfs_lock  * /
#define FLAG_SYSFS   4   / *  exported via/sys/class/gpio/control  * /
#define FLAG_TRIG_FALL   5   / *  trigger on falling edge  * /
#define FLAG_TRIG_RISE   6   / *  trigger on rising edge  * /
#define FLAG_ACTIVE_LOW   7   / *  sysfs value has active low  * /

#define ID_SHIFT   16   / *  add new flags before this one  * /

#define GPIO_FLAGS_MASK      ((1 << ID_SHIFT) - 1)
#define GPIO_TRIGGER_MASK(BIT(FLAG_TRIG_FALL) | BIT(FLAG_TRIG_RISE))

#ifdef CONFIG_DEBUG_FS
    const char           * label;
#endif
};
```

其中的 chip 表示该 GPIO 由哪个 gpio_chip 进行管理。其中，flags 表示该 GPIO 是否被请求、输入输出状态等与 GPIO 功能相关的状态及属性。

GPIO 框架中还需要对 gpio_chip 进行管理，主要是将实际的物理 GPIO 的管理实体 gpio_

chip 和逻辑层的 gpio_desc 进行绑定，相应绑定的 GPIO 号是在 gpio_chip 的 base 至 base + ngpio 之间。对之前 gpio_request 申请的，只有在绑定 gpio_chip 之后才能申请成功，继而对 GPIO 进行正确的操作。

具体的 gpio_chip 管理接口如下：

```
//加入 gpio_chip 并将其和所管理的 gpio_desc 进行绑定
extern int gpiochip_add(struct gpio_chip *chip);
//移除 gpio_chip 并对 gpio_desc 进行合适的操作移除绑定
extern int __must_check gpiochip_remove(struct gpio_chip *chip);
```

3. GPIO 的 sysfs 接口

针对应用程序对 GPIO 操作的需求，GPIO 库将 GPIO 开放到 sysfs 文件系统中，这样可以在用户层对 GPIO 进行相应的操作。在用户层进行操作的好处是，可以直接进行开发而不是通过驱动程序进行，另外从软件版权考虑可以将不同的版权更好的隔离避免彼此的污染。相应的接口如下：

```
//将 GPIO 输出给 sysfs 以供用户层使用
extern int gpio_export(unsigned gpio, bool direction_may_change);
//关闭 GPIO 在 sysfs 的接口
extern void gpio_unexport(unsigned gpio);
```

开放到 sysfs 的 GPIO 有如下的操作接口：

```
/*
 * /sys/class/gpio/gpioN... only for GPIOs that are exported
 *   /direction
 *       * MAY BE OMITTED if kernel won't allow direction changes
 *       * is read/write as "in" or "out"
 *       * may also be written as "high" or "low", initializing
 *         output value as specified("out" implies "low")
 *   /value
 *       * always readable, subject to hardware behavior
 *       * may be writable, as zero/nonzero
 *   /edge
 *       * configures behavior of poll(2)on/value
 *       * available only if pin can generate IRQs on input
 *       * is read/write as "none", "falling", "rising", or "both"
 *   /active_low
 *       * configures polarity of/value
 *       * is read/write as zero/nonzero
 *       * also affects existing and subsequent "falling" and "rising"
 *         /edge configuration
 */
```

以上是 GPIO 框架抽象层的实现，体系结构具体的实现主要就是实现并注册结构体 gpio_chip 及相关的操作。从整体上来说 GPIO 框架还是简单直接的。

4.8.3 TI 芯片 GPIO 管理相关实现详解

TI 芯片 GPIO 管理的具体实现仍是以 DM 3730 为例，关于 GPIO 模块本身，DM 81XX 等处理器与 DM 3730 是相同的，只是一些寄存器地址和参数不同。

先来看看硬件的框架和细节。DM 3730 GPIO 子系统框图如图 4-50 所示，引自《DM 3730 芯片手册》中第 3478 页框图。

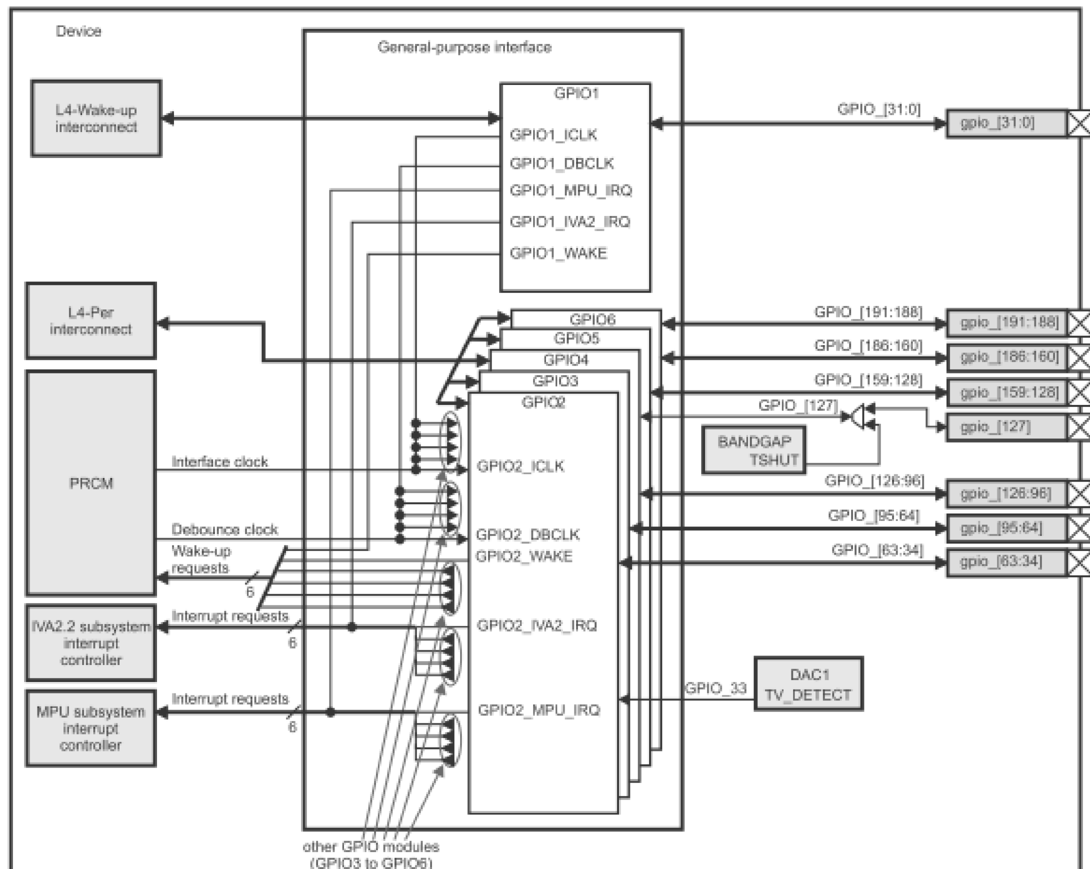


图 4-50 DM 3730 GPIO 子系统框图

从图 4-50 可见，DM 3730 一共有 6 组 GPIO 管理单元，每组管理 32 个 GPIO。其中 GPIO1 是比较特殊的，其原因在硬件电源管理技术中已提到，GPIO1 在 WKUP domain 中，而其他组的 GPIO 在其他的 power domain 中。WKUP domain 是特殊的 power domain，该 domain 的模块会一直供电。在每组 GPIO 中，除了标准的时钟之外，还有 debounce clock 用来进行 debounce 处理（主要在作为 input 时去抖，避免抖动产生的错误结果），另外还有唤醒系统的功能。GPIO 的中断信号同样可以提供给 ARM MPU 和 IVA DSP，从而不同的核都可以对外部的 GPIO 进行操作。

关于 GPIO 内部的系统设计如图 4-51 所示。图 4-51 引自《DM 3730 芯片手册》中第

3490 页框图。

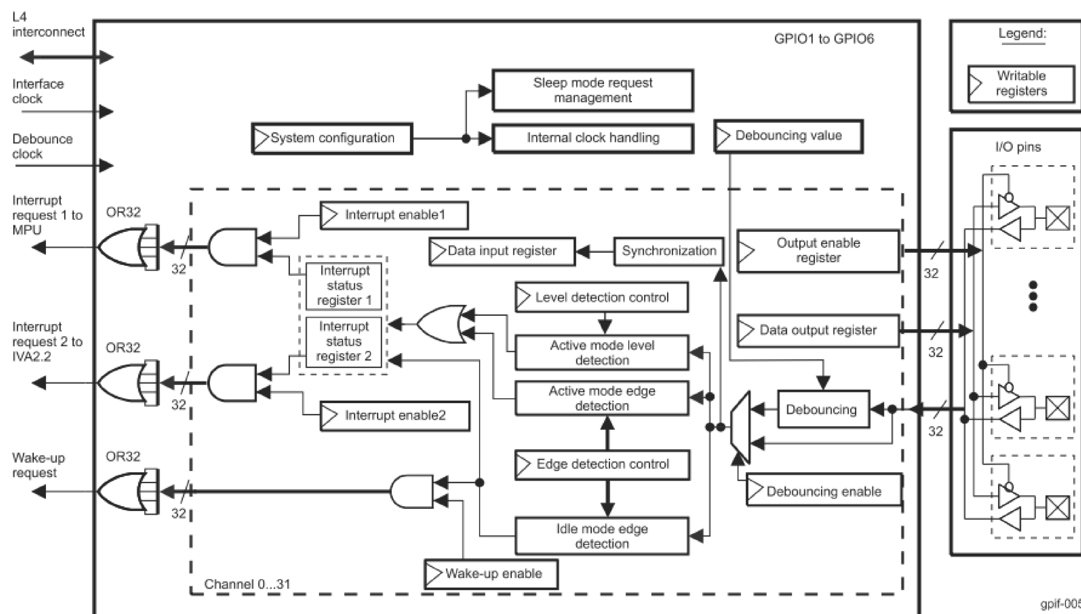


图 4-51 DM 3730 GPIO 内部系统设计图

从图 4-51 中可见，一组 GPIO 的设置寄存器接口，主要由系统控制、中断管理、去抖模块、输入输出特性等不同功能设置寄存器对相应的功能进行设置。

针对软件的实现，使用 GPIO lib 提供的管理实体 gpio_chip 不能满足 DM 3730 关于 GPIO 控制管理的需求，所以在芯片代码中对 gpio_chip 进行了封装并加入芯片特定的管理属性形成 gpio_bank 结构，详细内容如下：

```
struct gpio_bank {
    unsigned long pbase;
    void __iomem *base;
    u16 irq;
    u16 virtual_irq_start;
    int method;
#if defined(CONFIG_ARCH_OMAP16XX) || defined(CONFIG_ARCH_OMAP2PLUS)
    u32 suspend_wakeup;
    u32 saved_wakeup;
#endif
    u32 non_wakeup_gpios;
    u32 enabled_non_wakeup_gpios;

    u32 saved_datain;
    u32 saved_fallingdetect;
    u32 saved_risingdetect;
```

```
        u32 level_mask;
        u32 toggle_mask;
        spinlock_t lock;
        struct gpio_chip chip;
        struct clk  * dbck;
        u32 mod_usage;
        u32 dbck_enable_mask;
        struct device  * dev;
        bool dbck_flag;
        int stride;
    };
```

作为 DM 3730 芯片 GPIO 管理的核心数据结构，gpio_bank 中已经包含了 GPIO lib 需要的 gpio_chip，这样的实现符合面向对象的方法，从体系结构层次的角度考虑也是底层包含上层的内容。另外 gpio_bank 中还包含了用于 GPIO 控制的属性，这些属性会在具体的操作中用到。

下面介绍详细的实现。

1. 重要的初始化实现

对于 GPIO 功能芯片特殊的实现和 GPIO lib 的接口就是 gpio_chip。而芯片特殊实现本身也和填充 gpio_chip 以及注册 gpio_chip 相关，这些都是在初始化中实现的，了解这些初始化工作就可以大致了解 GPIO 芯片的特性。DM 3730 相关的初始化函数是 omap_gpio_chip_init，详细的内容和注释如下：

```
    static void __ init omap_gpio_chip_init(struct gpio_bank  * bank)
    {
        int j;
        static int gpio;

        //初始化该组 GPIO 的使用情况
        bank -> mod_usage = 0;
        / *
          * REVISIT eventually switch from OMAP - specific gpio structs
          * over to the generic ones
          * /
        //设置 gpio_chip 相关的接口主要是 GPIO 相关功能
        bank -> chip. request = omap_gpio_request;
        bank -> chip. free = omap_gpio_free;
        bank -> chip. direction_input = gpio_input;
        bank -> chip. get = gpio_get;
        bank -> chip. direction_output = gpio_output;
        bank -> chip. set_debounce = gpio_debounce;
```

```
        bank -> chip. set = gpio_set;
        bank -> chip. to_irq = gpio_2irq;
        if( bank_is_mpuio( bank) ) {
            //该分支是指 ARM MPU 中有 GPIO 控制器的情况。DM 3730 不是该情况
            bank -> chip. label = " mpuio" ;
#ifdef CONFIG_ARCH_OMAP16XX
            bank -> chip. dev = &omap_mpuio_device. dev;
#endif
            bank -> chip. base = OMAP_MPUIO(0) ;
        } else {
            //设置该组 GPIO 的起始 GPIO 号。对多组 GPIO 的情况需要按照顺序对该函数进行
            //调用
            bank -> chip. label = " gpio" ;
            bank -> chip. base = gpio;
            gpio += bank_width;
        }
        //设置该组 gpio_bank 管理的 GPIO 数目
        bank -> chip. ngpio = bank_width;

        //注册 gpio_chip 并和所管理的 gpio_desc 进行绑定
        gpiochip_add( &bank -> chip) ;

        //下面对一组 GPIO 所有的 GPIO 对应的中断处理进行初始化
        for( j = bank -> virtual_irq_start;
                 j < bank -> virtual_irq_start + bank_width; j ++ ) {
            //获得某个 GPIO 对应的中断描述符
            struct irq_desc  * d = irq_to_desc( j) ;

            lockdep_set_class( &d -> lock, &gpio_lock_class) ;
            //设置中断控制器的私有数据为该组 gpio_bank
            set_irq_chip_data( j, bank) ;
            //设置正确的中断控制器处理结构,DM 3730 是 gpio_irq_chip
            if( bank_is_mpuio( bank) )
                set_irq_chip( j, &mpuio_irq_chip) ;
            else
                set_irq_chip( j, &gpio_irq_chip) ;
            //设置简单的中断处理 handler,该 handler 需要外部进行中断控制器的处理
            set_irq_handler( j, handle_simple_irq) ;
            //设置该中断为无效状态
            set_irq_flags( j, IRQF_VALID) ;
        }
        //这里是针对中断级联的情况,之前是针对该组 GPIO 的每个 GPIO 的中断处理。而从系统的
```

```
        //角度不能为每个 gpio 都提供对 ARM 的中断信号,而是每组 GPIO 提供一个对 ARM 的中断
        //信号,这里需要对于该组 GPIO 对 ARM 的中断进行初始化。这里要设置中断为级联状态
        //及其 handler,保证该中断不能被其他模块申请。
        set_irq_chained_handler(bank -> irq, gpio_irq_handler);
        //设置中断处理的私有数据为当前 gpio_bank
        set_irq_data(bank -> irq, bank);
    }
```

分析了重要的初始化函数 omap_gpio_chip_init 之后，可见 GPIO 分为两个重要的功能组，一组是 gpio_chip 相关的功能接口，另一组是和中断相关的功能。下面分两部分进行详细的代码解析。

2. gpio_chip 功能接口

GPIO 的功能主要是通过 gpio_chip 的功能接口实现的。下面对相关实现代码进行详细解析：

```
    //请求获得特定 GPIO 的接口
    static int omap_gpio_request(struct gpio_chip *chip, unsigned offset)
    {
        //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
        struct gpio_bank *bank = container_of(chip, struct gpio_bank, chip);
        unsigned long flags;

        //由于该函数相当于在库函数中调用,所以使用高级别的锁保护
        spin_lock_irqsave(&bank -> lock, flags);

        /* Set trigger to none. You need to enable the desired trigger with
         * request_irq()or set_irq_type().
         */
        //功能接口就取消中断触发,在系统中断接口中进行相应的设置
        _set_gpio_triggering(bank, offset, IRQ_TYPE_NONE);

    …
        if(!cpu_class_is_omap1()){
            if(!bank -> mod_usage){
                //该组 GPIO 中申请的第一个 GPIO,则使能该模块
                void __iomem *reg = bank -> base;
                u32 ctrl;

                //先获得 control 寄存器的地址
                if(cpu_is_omap24xx() || cpu_is_omap34xx())
                    reg += OMAP24XX_GPIO_CTRL;
                else if(cpu_is_omap44xx() || cpu_is_ti816x())
```

```
                reg += OMAP4_GPIO_CTRL;
            ctrl = __raw_readl(reg);
            /* Module is enabled, clocks are not gated */
            //使能模块和时钟
            ctrl &= 0xFFFFFFFE;
            __raw_writel(ctrl, reg);
        }
        //标记该 GPIO 申请标识
        bank->mod_usage |= 1 << offset;
    }
    //释放锁
    spin_unlock_irqrestore(&bank->lock, flags);

    return 0;
}


//释放某个 GPIO 的接口
static void omap_gpio_free(struct gpio_chip *chip, unsigned offset)
{
    //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
    struct gpio_bank *bank = container_of(chip, struct gpio_bank, chip);
    unsigned long flags;

    //锁保护
    spin_lock_irqsave(&bank->lock, flags);
...
#if defined(CONFIG_ARCH_OMAP2) || defined(CONFIG_ARCH_OMAP3)
    if(bank->method == METHOD_GPIO_24XX) {
        //GPIO 要释放清除 GPIO 唤醒功能
        /* Disable wake-up during idle for dynamic tick */
        void __iomem *reg = bank->base + OMAP24XX_GPIO_CLEARWKUENA;
        __raw_writel(1 << offset, reg);
    }
#endif
#if defined(CONFIG_ARCH_OMAP4) || defined(CONFIG_ARCH_TI816X)
    if(bank->method == METHOD_GPIO_44XX) {
        /* Disable wake-up during idle for dynamic tick */
        void __iomem *reg = bank->base + OMAP4_GPIO_IRQWAKEN0;
    __raw_writel(1 << offset, reg);
}
#endif
```

```
        if(!cpu_class_is_omap1()){
            //标记 GPIO 释放
            bank->mod_usage &= ~(1 << offset);
            if(!bank->mod_usage){
                //该组 GPIO 都已经释放,则需要 disable 模块关闭 clock
                void __iomem *reg = bank->base;
                u32 ctrl;

                //获得 control 寄存器
                if(cpu_is_omap24xx() || cpu_is_omap34xx())
                    reg += OMAP24XX_GPIO_CTRL;
                else if(cpu_is_omap44xx() || cpu_is_ti816x())
                    reg += OMAP4_GPIO_CTRL;
                ctrl = __raw_readl(reg);
                /* Module is disabled, clocks are gated */
                //disable 该模块关闭 clock
                ctrl |= 1;
                __raw_writel(ctrl, reg);
            }
        }
        //由于 GPIO 可能做中断,所以在释放时需要清除中断相关状态等操作
        _reset_gpio(bank, bank->chip.base + offset);
        //释放锁
        spin_unlock_irqrestore(&bank->lock, flags);
    }

    //设置 GPIO 为输入的接口
    static int gpio_input(struct gpio_chip *chip, unsigned offset)
    {
        struct gpio_bank *bank;
        unsigned long flags;

        //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
        bank = container_of(chip, struct gpio_bank, chip);
        //锁保护
        spin_lock_irqsave(&bank->lock, flags);
        //设置方向为输入
        _set_gpio_direction(bank, offset, 1);
        //释放锁
        spin_unlock_irqrestore(&bank->lock, flags);
        return 0;
    }
```

```
//获得 GPIO 信号的值
static int gpio_get(struct gpio_chip  * chip, unsigned offset)
{
    struct gpio_bank  * bank;
    void __ iomem  * reg;
    int gpio;
    u32 mask;

    //获得 GPIO 表示方向状态寄存器
    gpio = chip -> base + offset;
    bank = get_gpio_bank(gpio);
    reg = bank -> base;
    mask = 1 << get_gpio_index(gpio);

    //根据方向状态获得 GPIO 的值
    if(gpio_is_input(bank, mask))
        return _get_gpio_datain(bank, gpio);
    else
        return _get_gpio_dataout(bank, gpio);
}
//设置 GPIO 为输出并设置输出值
static int gpio_output(struct gpio_chip  * chip, unsigned offset, int value)
{
    struct gpio_bank  * bank;
    unsigned long flags;

    //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
    bank = container_of(chip, struct gpio_bank, chip);
    //锁保护
    spin_lock_irqsave(&bank -> lock, flags);
    //设置输出值
    _set_gpio_dataout(bank, offset, value);
    //设置 GPIO 方向为输出
    _set_gpio_direction(bank, offset, 0);
    //释放锁
    spin_unlock_irqrestore(&bank -> lock, flags);
    return 0;
}

//设置去抖模块
static int gpio_debounce(struct gpio_chip  * chip, unsigned offset,
```

```
            unsigned debounce)
{
    struct gpio_bank  * bank;
    unsigned long flags;

    //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
    bank = container_of(chip, struct gpio_bank, chip);

    if(!bank -> dbck){
        //如果没有获得去抖模块相关时钟,则首先获得并打开相关时钟
        bank -> dbck = clk_get(bank -> dev, "dbclk");
        if(IS_ERR(bank -> dbck))
            dev_err(bank -> dev, "Could not get gpio dbck\n");
    }

    //获得锁
    spin_lock_irqsave(&bank -> lock, flags);
    //设置去抖功能及参数
    _set_gpio_debounce(bank, offset, debounce);
    //释放锁
    spin_unlock_irqrestore(&bank -> lock, flags);

    return 0;
}

//设置 GPIO 的值
static void gpio_set(struct gpio_chip  * chip, unsigned offset, int value)
{
    struct gpio_bank  * bank;
    unsigned long flags;

    //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
    bank = container_of(chip, struct gpio_bank, chip);
    //获得锁
    spin_lock_irqsave(&bank -> lock, flags);
    //设置 GPIO 输出值
    _set_gpio_dataout(bank, offset, value);
    //释放锁
    spin_unlock_irqrestore(&bank -> lock, flags);
}

//获得 GPIO 对应的中断号
```

```
static int gpio_2irq(struct gpio_chip *chip, unsigned offset)
{
    struct gpio_bank *bank;

    //首先获得该组 GPIO 的芯片级管理实体 gpio_bank
    //该组 GPIO 的起始中断号加上该 GPIO 的偏移就是 GPIO 对应的中断号
    bank = container_of(chip, struct gpio_bank, chip);
    return bank->virtual_irq_start + offset;
}
```

对 GPIO 相关功能的芯片特殊实现，主要进行了函数功能的分析，具体到子函数的细节可以查看相关代码。

3. GPIO 中断相关实现

GPIO 中断相关的实现，最主要的是中断控制器相关的接口实现以及中断级联中 GPIO 组的中断处理函数。

先来看看 GPIO 的中断控制器相关接口。这里实际实现了级联的中断处理。

```
static struct irq_chip gpio_irq_chip = {
    .name       = "GPIO",
    .shutdown   = gpio_irq_shutdown,
    .ack    = gpio_ack_irq,
    .mask       = gpio_mask_irq,
    .unmask     = gpio_unmask_irq,
    .set_type = gpio_irq_type,
    .set_wake = gpio_wake_enable,
};
```

该中断控制器负责一组 GPIO 的中断处理，详细的接口代码分析如下：

```
//关掉中断
static void gpio_irq_shutdown(unsigned int irq)
{
    //首先根据中断号获得 GPIO 值
    unsigned int gpio = irq - IH_GPIO_BASE;
    //找到对应的 GPIO 组
    struct gpio_bank *bank = get_irq_chip_data(irq);

    //清除中断状态关闭相应 GPIO 中断
    _reset_gpio(bank, gpio);
}

//对中断控制器确认中断处理以便可以接收新的中断
```

```
static void gpio_ack_irq(unsigned int irq)
{
    //首先根据中断号获得 GPIO 值
    unsigned int gpio = irq - IH_GPIO_BASE;
    //找到对应的 GPIO 组
    struct gpio_bank *bank = get_irq_chip_data(irq);

    //清楚中断状态位
    _clear_gpio_irqstatus(bank, gpio);
}

//屏蔽中断
static void gpio_mask_irq(unsigned int irq)
{
    //首先根据中断号获得 GPIO 值
    unsigned int gpio = irq - IH_GPIO_BASE;
    //找到对应的 GPIO 组
    struct gpio_bank *bank = get_irq_chip_data(irq);

    //disable 相应中断
    _set_gpio_irqenable(bank, gpio, 0);
    //关闭触发功能
    _set_gpio_triggering(bank, get_gpio_index(gpio), IRQ_TYPE_NONE);
}

//开放中断功能
static void gpio_unmask_irq(unsigned int irq)
{
    //首先根据中断号获得 GPIO 值
    unsigned int gpio = irq - IH_GPIO_BASE;
    //找到对应的 GPIO 组
    struct gpio_bank *bank = get_irq_chip_data(irq);
    unsigned int irq_mask = 1 << get_gpio_index(gpio);
    //获得中断描述符并检测触发方式
    struct irq_desc *desc = irq_to_desc(irq);
    u32 trigger = desc->status & IRQ_TYPE_SENSE_MASK;

    //如果之前已经设置触发方式则设置触发
    if(trigger)
    _set_gpio_triggering(bank, get_gpio_index(gpio), trigger);

    /* For level-triggered GPIOs, the clearing must be done after
```

```
     * the HW source is cleared, thus after the handler has run */
    //电平触发的中断需要先清除具体的 GPIO 中断状态
    if(bank->level_mask & irq_mask) {
        _set_gpio_irqenable(bank, gpio, 0);
        _clear_gpio_irqstatus(bank, gpio);
    }

    //使能 GPIO 中断
    _set_gpio_irqenable(bank, gpio, 1);
}

//设置中断触发类型的接口
static int gpio_irq_type(unsigned irq, unsigned type)
{
    struct gpio_bank *bank;
    unsigned gpio;
    int retval;
    unsigned long flags;

    //获得 GPIO 号
    if(!cpu_class_is_omap2() && irq > IH_MPUIO_BASE)
        gpio = OMAP_MPUIO(irq - IH_MPUIO_BASE);
    else
        gpio = irq - IH_GPIO_BASE;

    if(check_gpio(gpio) < 0)
        return -EINVAL;

    if(type & ~IRQ_TYPE_SENSE_MASK)
        return -EINVAL;

    /* OMAP1 allows only only edge triggering */
    if(!cpu_class_is_omap2()
        &&(type &(IRQ_TYPE_LEVEL_LOW | IRQ_TYPE_LEVEL_HIGH)))
      return -EINVAL;

    //获得 GPIO 所在的组管理实体
    bank = get_irq_chip_data(irq);
    //获得 GPIO 组管理实体锁
    spin_lock_irqsave(&bank->lock, flags);
    //设置中断触发方式
    retval = _set_gpio_triggering(bank, get_gpio_index(gpio), type);
```

```
        if(retval ==0) {
            struct irq_desc *d = irq_to_desc(irq);

            d->status &= ~IRQ_TYPE_SENSE_MASK;
            d->status |= type;
        }
        //释放 GPIO 组管理实体锁
        spin_unlock_irqrestore(&bank->lock, flags);

        //根据中断触发方式设置合适的中断处理 handler,这些 handler 由 ARM 体系结构
        //代码统一提供,属于中断触发处理的标准 handler,中断处理章节介绍差异
        if(type &(IRQ_TYPE_LEVEL_LOW | IRQ_TYPE_LEVEL_HIGH))
            __set_irq_handler_unlocked(irq, handle_level_irq);
        else if(type &(IRQ_TYPE_EDGE_FALLING | IRQ_TYPE_EDGE_RISING))
            __set_irq_handler_unlocked(irq, handle_edge_irq);

        return retval;
    }

    //GPIO 唤醒功能设定接口
    static int gpio_wake_enable(unsigned int irq, unsigned int enable)
    {
        //首先根据中断号获得 GPIO 值
        unsigned int gpio = irq - IH_GPIO_BASE;
        struct gpio_bank *bank;
        int retval;

        if(check_gpio(gpio) < 0)
            return -ENODEV;
        //获得 GPIO 组管理实体
        bank = get_irq_chip_data(irq);
        //根据 enable 参数设置 GPIO 唤醒功能
        retval = _set_gpio_wakeup(bank, get_gpio_index(gpio), enable);

        return retval;
    }
```

从中可见，GPIO 不仅有基本的中断操作，还有唤醒系统的能力，所以提供了中断控制器中 set_wake 的接口（该接口是老的接口，但是可以通过新的接口 irq_set_wake 转换调用，内核中断处理框架 irq_chip_set_defaults 会建立这种转换）。

接下来解析一下级联的中断处理，由于 ARM 无法获得一组 GPIO 中具体哪个 GPIO 上报的中断，所以需要通过该级联中断处理，进行具体的解析并需要对具体的 GPIO 中断进行处理。

```
//级联的中断处理函数
static void gpio_irq_handler(unsigned int irq, struct irq_desc *desc)
{
    void __iomem *isr_reg = NULL;
    u32 isr;
    unsigned int gpio_irq, gpio_index;
    struct gpio_bank *bank;
    u32 retrigger = 0;
    int unmasked = 0;

    //先对上层的中断控制器进行 ack 操作,主要是由于这里需要进行多个
    //中断的处理
    desc->chip->ack(irq);

    //获得 GPIO 组管理实体
    bank = get_irq_data(irq);
...
    //获得正确的中断状态寄存器地址
#if defined(CONFIG_ARCH_OMAP2) || defined(CONFIG_ARCH_OMAP3)
    if(bank->method == METHOD_GPIO_24XX)
        isr_reg = bank->base + OMAP24XX_GPIO_IRQSTATUS1;
#endif
#if defined(CONFIG_ARCH_OMAP4) || defined(CONFIG_ARCH_TI816X)
    if(bank->method == METHOD_GPIO_44XX)
        isr_reg = bank->base + OMAP4_GPIO_IRQSTATUS0;
#endif

    if(WARN_ON(!isr_reg))
        goto exit;

    //这里是循环检查直到所有需要处理的中断处理完
    while(1) {
        u32 isr_saved, level_mask = 0;
        u32 enabled;

        //检查哪些 GPIO 的中断 enable
        enabled = _get_gpio_irqbank_mask(bank);
        //查看哪些 GPIO 上报了中断
        isr_saved = isr = __raw_readl(isr_reg) & enabled;

        if(cpu_is_omap15xx() &&(bank->method == METHOD_MPUIO))
```

```
        isr & = 0x0000ffff;

    //检查哪些是电平触发中断
    if( cpu_class_is_omap2( ) ) {
        level_mask = bank -> level_mask & enabled;
    }

    /* clear edge sensitive interrupts before handler(s) are
    called so that we don't miss any interrupt occurred while
    executing them */
    //对于边沿触发的中断现在可以先清除,这样可以避免错过中断
    _enable_gpio_irqbank( bank, isr_saved & ~level_mask, 0);
    _clear_gpio_irqbank( bank, isr_saved & ~level_mask);
    _enable_gpio_irqbank( bank, isr_saved & ~level_mask, 1);

    /* if there is only edge sensitive GPIO pin interrupts
    configured, we could unmask GPIO bank interrupt immediately */
    //所有的 GPIO 都是边沿触发模式,并且上一级中断控制器还没有 umask
    //相应的中断,这里可以 umask 相应中断,并标记已经 umask
    if(! level_mask && ! unmasked) {
        unmasked = 1;
        desc -> chip -> unmask( irq);
    }

    //这里的 retrigger 原意是标记在处理过程中发生的中断,但是实际并没有
    //使用该值
    isr | = retrigger;
    retrigger = 0;
    //没有中断上报的情况就退出循环
    if(! isr)
        break;

    //这里需要获得实际的 GPIO 中断并处理
    //首先获得该组 GPIO 的初始中断号
    gpio_irq = bank -> virtual_irq_start;
    //遍历所有触发的中断
    for( ; isr ! = 0; isr >> = 1, gpio_irq ++ ) {
        gpio_index = get_gpio_index( irq_to_gpio( gpio_irq) );

        //没有触发中断的 GPIO 跳过
        if(! (isr & 1) )
            continue;
```

```
…

                //处理触发的 GPIO 中断,此处为实际的 GPIO 中断,相当于二级中断
                generic_handle_irq(gpio_irq);
            }
        }
        /* if bank has any level sensitive GPIO pin interrupt
        configured, we must unmask the bank interrupt only after
        handler(s) are executed in order to avoid spurious bank
        interrupt */
        //这里如果没有 umask,说明有电平触发的中断,需要在此时进行 umask 操作
exit:
        if(! unmasked)
            desc->chip->unmask(irq);

}
```

至此，关于 GPIO 的中断部分主要框架就分析完了。中断处理还要注意电平触发和边沿触发的流程是不同的，这些都要体现在级联的中断处理中。

4.9 引脚复用（pin mux）

引脚复用一直是内核代码所忽视的内容，而对 SoC 来说又十分重要。SoC 中包含了很多功能模块，而由于芯片面积的限制，其引脚数目也是受限的，不可能所有功能模块都有单独的引脚，这就要求有引脚复用。

4.9.1 引脚复用的基本需求

引脚通常是和使用者相关的，相应的使用者是某些功能模块。要使得功能模块能够找到其对应的引脚并进行正确的设置，就需要引脚复用模块对每个引脚及其能被哪些模块使用进行管理。这其中的方法可以根据需要自行设计，但是该功能是引脚复用必须实现的。

另外即使是同一款芯片针对不同的应用场合会有不同的封装方式，而对引脚来说不同的封装就会意味着引脚数目的差别，这样对引脚复用来说，不同的封装实际的管理数据又会有差别，所以引脚复用还要和封装方式结合。这些都要体现在实际的系统中。

最后引脚复用还需要执行时能够对引脚属性以及功能进行配置，以实现灵活的功能，另外电源管理也是需要该功能的。

4.9.2 引脚复用框架介绍

关于引脚复用框架，一直以来 Linux 内核并没有提供统一的功能框架，此功能也只能由芯片厂商来完成了，直到最近几版内核才采用 STE 提出的 pin control 架构，从而显现出融合的趋势。其相应的框架如图 4-52 所示。

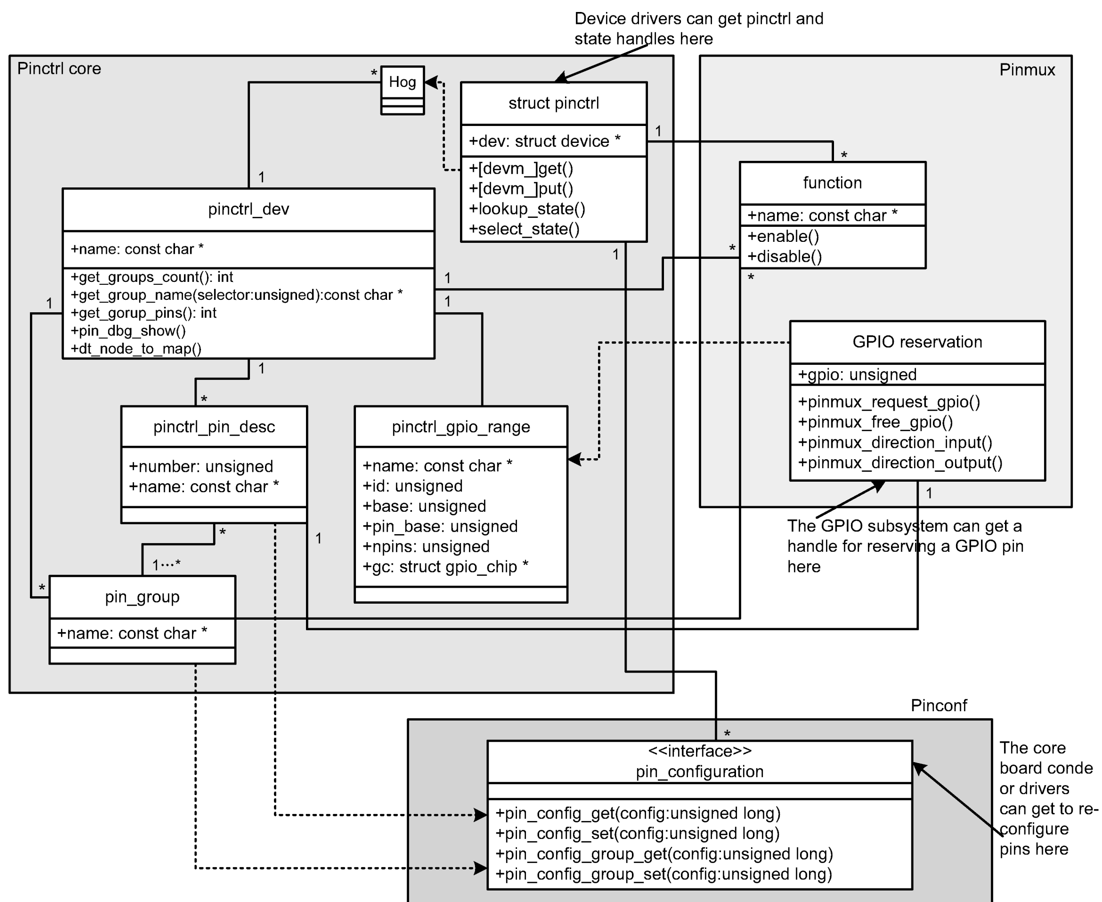


图 4-52　pin control 系统框架

从图 4-52 中可见，pin control 架构对使用引脚的设备、引脚的分组、引脚的描述、管脚的设置都进行了抽象，再将它们进行合理的组合，并与具体的设备模块结合进行管理，形成统一的引脚复用管理框架。

由于 DM 3730 等 TI 芯片在实现中还没有出现 pin control 架构，而是使用 TI 自己设计的引脚复用架构。后续是以芯片为主进行介绍，所以对于 pin control 的设计与实现就不进行深入地介绍了，其基本的思路还是和 TI 的引脚复用架构类似，读者可以深入了解 TI 引脚复用实现，然后继续深入了解 pin control 的架构。

4.9.3　TI 芯片引脚复用相关实现详解

1. 硬件引脚控制寄存器

首先来看看硬件上引脚复用提供给软件的接口寄存器，如图 4-53 所示。图 4-53 引自《DM3730 芯片手册》中第 2449 的表。

图 4-53 中引用的只是 DM3730 引脚控制相关寄存器的一部分，从中可见对于某个固定的引脚可能会有不同的模式，而不同的模式表示该引脚为不同的功能模块提供服务，特殊的是 safe_mode 并不为具体的功能模块服务。从硬件的角度，引脚为 safe_mode 时，即使打开

Register Name	Physical Address	Mode0	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
CONTROL_PADCONF_SDRC_CLK[31:16]	0x4800 2070	sdrc_dqs0							
CONTROL_PADCONF_SDRC_DQS1[15:0]	0x4800 2074	sdrc_dqs1							
CONTROL_PADCONF_SDRC_DQS1[31:16]	0x4800 2074	sdrc_dqs2							
CONTROL_PADCONF_SDRC_DQS3[15:0]	0x4800 2078	sdrc_dqs3							
CONTROL_PADCONF_SDRC_DQS3[31:16]	0x4800 2078	gpmc_a1				gpio_34			safe_mode
CONTROL_PADCONF_GPMC_A2[15:0]	0x4800 207C	gpmc_a2				gpio_35			safe_mode
CONTROL_PADCONF_GPMC_A2[31:16]	0x4800 207C	gpmc_a3				gpio_36			safe_mode
CONTROL_PADCONF_GPMC_A4[15:0]	0x4800 2080	gpmc_a4				gpio_37			safe_mode
CONTROL_PADCONF_GPMC_A4[31:16]	0x4800 2080	gpmc_a5				gpio_38			safe_mode
CONTROL_PADCONF_GPMC_A6[15:0]	0x4800 2084	gpmc_a6				gpio_39			safe_mode
CONTROL_PADCONF_GPMC_A6[31:16]	0x4800 2084	gpmc_a7				gpio_40			safe_mode
CONTROL_PADCONF_GPMC_A8[15:0]	0x4800 2088	gpmc_a8				gpio_41			safe_mode
CONTROL_PADCONF_GPMC_A8[31:16]	0x4800 2088	gpmc_a9	sys_ndmareq2			gpio_42			safe_mode
CONTROL_PADCONF_GPMC_A10[15:0]	0x4800 208C	gpmc_a10	sys_ndmareq3			gpio_43			safe_mode
CONTROL_PADCONF_GPMC_A10[31:16]	0x4800 208C	gpmc_d0							
CONTROL_PADCONF_GPMC_D1[15:0]	0x4800 2090	gpmc_d1							
CONTROL_PADCONF_GPMC_D1[31:16]	0x4800 2090	gpmc_d2							
CONTROL_PADCONF_GPMC_D3[15:0]	0x4800 2094	gpmc_d3							
CONTROL_PADCONF_GPMC_D3[31:16]	0x4800 2094	gpmc_d4							
CONTROL_PADCONF_GPMC_D5[15:0]	0x4800 2098	gpmc_d5							
CONTROL_PADCONF_GPMC_D5[31:16]	0x4800 2098	gpmc_d6							
CONTROL_PADCONF_GPMC_D7[15:0]	0x4800 209C	gpmc_d7							
CONTROL_PADCONF_GPMC_D7[31:16]	0x4800 209C	gpmc_d8				gpio_44			safe_mode
CONTROL_PADCONF_GPMC_D9[15:0]	0x4800 20A0	gpmc_d9				gpio_45			safe_mode
CONTROL_PADCONF_GPMC_D9[31:16]	0x4800 20A0	gpmc_d10				gpio_46			safe_mode
CONTROL_PADCONF_GPMC_D11[15:0]	0x4800 20A4	gpmc_d11				gpio_47			safe_mode
CONTROL_PADCONF_GPMC_D11[31:16]	0x4800 20A4	gpmc_d12				gpio_48			safe_mode
CONTROL_PADCONF_GPMC_D13[15:0]	0x4800 20A8	gpmc_d13				gpio_49			safe_mode
CONTROL_PADCONF_GPMC_D13[31:16]	0x4800 20A8	gpmc_d14				gpio_50			safe_mode
CONTROL_PADCONF_GPMC_D15[15:0]	0x4800 20AC	gpmc_d15				gpio_51			safe_mode
CONTROL_PADCONF_GPMC_D15[31:16]	0x4800 20AC	gpmc_ncs0							
CONTROL_PADCONF_GPMC_NCS1[15:0]	0x4800 20B0	gpmc_ncs1				gpio_52			safe_mode

图 4-53　DM3730 引脚复用相关寄存器

输入功能硬件也不会出现误传信号的问题。

引脚配置寄存器的设置如图 4-54 所示。图 4-54 引自《DM 3730 芯片手册》中第 2444 页框图。

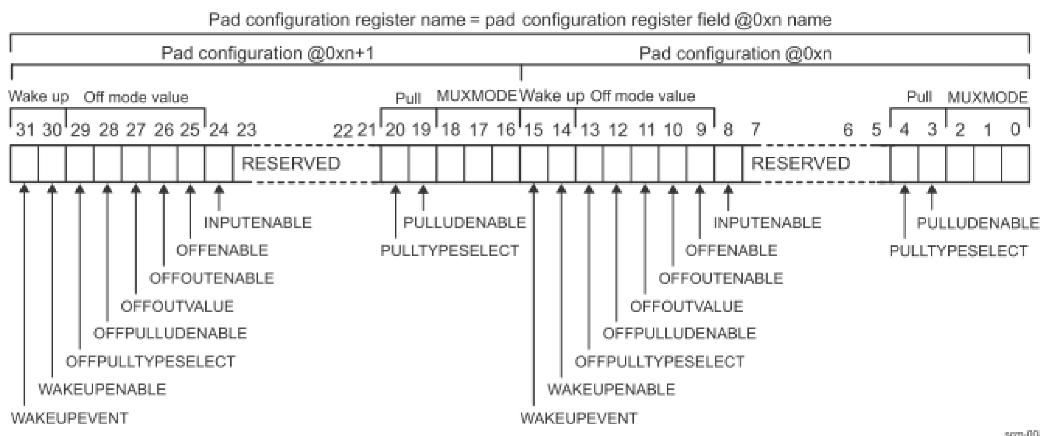


图 4-54　DM 3730 引脚配置寄存器设置

从图 4-54 中可见，引脚的设置除了具体针对模块功能之外，还有不同的引脚属性配置（比如上下拉，以及电源管理的属性）。从这些设置的内容可见，引脚复用属于横切功能，所以从设计的角度还是要考虑对于不同设备的支持。这就要求数据结构设计合理，并且要有通用性。

2. 引脚复用管理数据结构

下面看看 TI 的引脚复用的重要数据结构和详细说明。

```
/**
 * struct omap_device_pad - device specific pad configuration
 * @name:signal name
 * @flags:pad specific runtime flags
 * @enable:runtime value for a pad
 * @idle:idle value for a pad
 * @off:off value for a pad, defaults to safe mode
 * @partition:mux partition
 * @mux:mux register
 */
//某个设备模块的某个引脚的属性
struct omap_device_pad {
	char			*name;
	//状态标识
	u8			flags;
	//模块 enable 时的值
	u16			enable;
	//模块 idle 时的值
	u16			idle;
	//模块 off 时的值
	u16			off;
	//所属的引脚管理组
	struct omap_mux_partition *partition;
	//具体的引脚控制实体
	struct omap_mux *mux;
};

/**
  * struct omap_board_data - board specific device data
  * @id: instance id
  * @flags: additional flags for platform init code
  * @pads: array of device specific pads
  * @pads_cnt: ARRAY_SIZE() of pads
  */
//板级某个设备使用引脚的初始化数据
struct omap_board_data {
	//设备号,一类设备的 ID 号
	int			id;
	u32			flags;
	//该设备所有引脚的数据
	struct omap_device_pad *pads;
	//引脚数目
```

```
    in          tpads_cnt;
};

/**
 * @name: name of the current partition
 * @flags: flags specific to this partition
 * @phys: physical address
 * @size: partition size
 * @base: virtual address after ioremap
 * @muxmodes: list of nodes that belong to a partition
 * @node: list node for the partitions linked list
 */
//这里是控制一组引脚的结构,引脚控制寄存器在连续的物理空间为一组
struct omap_mux_partition {
    const char                  *name;
    u32 flags;
    //寄存器物理地址
    u32 phys;
    u32 size;
    //映射地址
    void __iomem *base;
    //链表连接所有的引脚控制结构 omap_mux_entry,其中包含 omap_mux
    struct list_headmuxmodes;
    //链表连接多个管理引脚的控制组
    struct list_headnode;
};
/**
 * struct omap_mux - data for omap mux register offset and it's value
 * @reg_offset:mux register offset from the mux base
 * @gpio:GPIO number
 * @muxnames:available signal modes for a ball
 * @balls:available balls on the package
 * @partition:mux partition
 */
//具体的控制某个引脚的设置
struct omap_mux {
    //在一组 padconf 寄存器中的偏移值
    u16   reg_offset;
    //该寄存器对应的 GPIO 值
    u16   gpio;
#ifdef CONFIG_OMAP_MUX
    char *muxnames[OMAP_MUX_NR_MODES];
```

```
#ifdef CONFIG_DEBUG_FS
       char * balls[OMAP_MUX_NR_SIDES];
#endif
#endif
};

/**
 * struct omap_board_mux - data for initializing mux registers
 * @reg_offset:mux register offset from the mux base
 * @mux_value:desired mux value to set
 */
//初始化时使用的结构用于特定引脚的属性设置
struct omap_board_mux {
       u16 reg_offset;
       u16 value;
};

//下面是 omap_device_pad 中 flag 的不同标识位的含义
#define OMAP_DEVICE_PAD_ENABLED       BIT(7)   /* Not needed for board - *.c */
#define OMAP_DEVICE_PAD_REMUX         BIT(1)   /* Dynamically remux a pad,
                                   needs enable, idle and off
                                   values */
#define OMAP_DEVICE_PAD_WAKEUP        BIT(0)   /* Pad is wake - up capable */
```

3. 引脚复用管理相关函数

引脚的管理分为整体和设备模块相关两部分。先来看看整体的引脚管理，主要形成 omap_mux_partition 的信息，具体的代码与分析如下：

```
//该函数在 machine_desc 中的 init_machine 调用
//主要是对每个连续的 partition 的设置时会调用. 进而产生相应
//的 partition 管理信息
int __init omap_mux_init(const char *name, u32 flags,
              u32 mux_pbase, u32 mux_size,
              struct omap_mux  *superset,
              struct omap_mux  *package_subset,
              struct omap_board_mux  *board_mux,
              struct omap_ball  *package_balls)
{
       struct omap_mux_partition  *partition;

       //分配 partition 管理结构
       partition = kzalloc(sizeof(struct omap_mux_partition), GFP_KERNEL);
       if(! partition)
```

```
        return -ENOMEM;

    //对于 partition 管理结构赋值
    partition->name = name;
    //该 flag 主要为
    //#define OMAP_MUX_REG_8BIT(1 << 0)
    //#define OMAP_MUX_GPIO_IN_MODE3(1 << 1)
    partition->flags = flags;
    //size 为有多少 pin mux 配置属性,主要是寄存器数量
    partition->size = mux_size;
    //寄存器物理基地址
    partition->phys = mux_pbase;
    //获得虚拟地址,主要为后续的 read/write 操作提供虚拟地址
    partition->base = ioremap(mux_pbase, mux_size);
    if(! partition->base) {
        pr_err("%s: Could not ioremap mux partition at 0x%08x\n",
            __func__, partition->phys);
        return -ENODEV;
    }

    //初始化该 partition 的 pin mux 链表
    INIT_LIST_HEAD(&partition->muxmodes);

    //每次调用该函数管理 partition 的链表都会增加
    list_add_tail(&partition->node, &mux_partitions);
    mux_partitions_cnt++;
    pr_info("%s: Add partition: #%d: %s, flags: %x\n", __func__,
        mux_partitions_cnt, partition->name, partition->flags);

    //对封装进行检查必要时生成 debugfs 的信息
    omap_mux_init_package(superset, package_subset, package_balls);
    //实例化 partition 的 pin mux 链表,包括 DM 3730 所有引脚的属性
    omap_mux_init_list(partition, superset);
    //板级信号的特别配置
    omap_mux_init_signals(partition, board_mux);

    return 0;
}
```

这些工作主要是在板级初始化的时候做。板级初始化阶段就可以进行特定的引脚设置工作，相应的引脚参数是由特定的 omap_board_mux 来进行设置，在 board-omap3evm. c 中进行相关设置的代码如下：

```
static struct omap_board_mux omap35x_board_mux[ ] __initdata = {
    OMAP3_MUX(SYS_NIRQ, OMAP_MUX_MODE0 | OMAP_PIN_INPUT_PULLUP |
        OMAP_PIN_OFF_INPUT_PULLUP | OMAP_PIN_OFF_OUTPUT_LOW |
        OMAP_PIN_OFF_WAKEUPENABLE),
...
    { .reg_offset = OMAP_MUX_TERMINATOR },
};
```

与设备相关的引脚管理的接口函数和详细分析如下：

```
//此变量为存放 bootargs 中的 mux 的配置说明
//通过 omap_mux = 来设置.
static char *omap_mux_options;

//omap3 所有的 GPIO 都是通过 mode4 来配置的,并且 GPIO 号在不同的 pin
//没有重复
//该函数主要是对相应的 GPIO pin 进行设置,设置成 GPIO 使用,将 pin mux
//配置成 val 的值,val 为 pin 的 inputenable、chip off 和 wakeup 时的属性
//该功能通常在 board 的相应模块初始化时对所使用的 GPIO pin 进行配置
static int __init _omap_mux_init_gpio(struct omap_mux_partition *partition,
                        int gpio, int val)
{
    struct omap_mux_entry *e;
    struct omap_mux *gpio_mux = NULL;
    u16 old_mode;
    u16 mux_mode;
    int found = 0;

    //首先获得该组 pin mux 的链表,其中是 omap_mux
    struct list_head *muxmodes = &partition->muxmodes;

    if(! gpio)
        return -EINVAL;

    //查找相应的 GPIO pin 的配置,并记录查找到的个数
    list_for_each_entry(e, muxmodes, node) {
        struct omap_mux *m = &e->mux;
        if(gpio == m->gpio) {
            gpio_mux = m;
            found++;
        }
```

```
        }

        //如果没有找到或者找到的个数多于 1 个都是错的.
        if(found == 0) {
            pr_err("%s: Could not set gpio%i\n", __func__, gpio);
            return -ENODEV;
        }

        if(found > 1) {
            pr_info("%s: Multiple gpio paths(%d) for gpio%i\n", __func__,
                found, gpio);
            return -EINVAL;
        }

        //读取相应的 pin 的配置
        old_mode = omap_mux_read(partition, gpio_mux->reg_offset);
        //重新设置 pin 的 mode 为 GPIO,omap3 上 GPIO 的 mode 为 4.
        //其他属性不修改.
        mux_mode = val & ~(OMAP_MUX_NR_MODES - 1);
        if(partition->flags & OMAP_MUX_GPIO_IN_MODE3)
            mux_mode |= OMAP_MUX_MODE3;
        else
            mux_mode |= OMAP_MUX_MODE4;
        pr_debug("%s: Setting signal %s.gpio%i 0x%04x -> 0x%04x\n", __func__,
            gpio_mux->muxnames[0], gpio, old_mode, mux_mode);
        omap_mux_write(partition, mux_mode, gpio_mux->reg_offset);

        return 0;
}

//对其他模块的设置 pin 为 GPIO 的接口
//由于每个 GPIO 只会有一个引脚与之相对应,GPIO 又需要根据 GPIO 号进行
//灵活简单的设置,所以为 GPIO 单独提供设置的接口
int __init omap_mux_init_gpio(int gpio, int val)
{
        struct omap_mux_partition *partition;
        int ret;

        //遍历所有的 partition 查找相应的 GPIO pin
        //并将相应的 pin 配置成 gpio 模式.
        list_for_each_entry(partition, &mux_partitions, node) {
            ret = _omap_mux_init_gpio(partition, gpio, val);
```

```
        if(! ret)
            return ret;
    }

    return -ENODEV;
}

//该函数通过 mode0. muxmode format 的格式查找相应的 pin mux 属性
//通常传入的参数只有 mode0,mode0 是指相应的 pin 在 mode0 时的名字.
static int __init _omap_mux_get_by_name(struct omap_mux_partition *partition,
                                        const char *muxname,
                                        struct omap_mux **found_mux)
{
    struct omap_mux *mux = NULL;
    struct omap_mux_entry *e;
    const char *mode_name;
    int found = 0, found_mode, mode0_len = 0;

    //先获得 partition 中的 pin mux 列表.
    struct list_head *muxmodes = &partition->muxmodes;

    //获得 mode0. muxmode 中的 muxmode 的名字.
    //如果没有 muxmode 则直接取 mode0
    mode_name = strchr(muxname,'.');
    if(mode_name) {
        mode0_len = strlen(muxname) - strlen(mode_name);
        mode_name++;
    } else {
        mode_name = muxname;
    }

    //遍历 mux pin 的链表,查找相应的 pin mux 属性
    list_for_each_entry(e, muxmodes, node) {
        char *m0_entry;
        int i;

        mux = &e->mux;
        m0_entry = mux->muxnames[0];

        //首先检查 mode0 是否匹配
        /* First check for full name in mode0. muxmode format */
        if(mode0_len && strncmp(muxname, m0_entry, mode0_len))
```

```
                continue;

            /* Then check for muxmode only */
            //这里检查 muxmode 是否匹配,之前如果
            //没有 muxmode,则 muxmode 为 mode0,这里相当于重
            //新比较一下 mode0,则一定匹配
            for(i = 0; i < OMAP_MUX_NR_MODES; i ++ ) {
                char *mode_cur = mux -> muxnames[i];

                if(! mode_cur)
                    continue;

                if(! strcmp(mode_name, mode_cur)) {
                    *found_mux = mux;
                    found ++ ;
                    found_mode = i;
                }
            }
        }

        if(found == 1) {
            //返回找到的 pin mux 的 mode 属性,通常为 0
            return found_mode;
        }

        //如果找到多于 1 个则报错.
        if(found > 1) {
            pr_err("%s: Multiple signal paths(%i) for %s\n", __func__,
                    found, muxname);
        return -EINVAL;
        }

        pr_err("%s: Could not find signal %s\n", __func__, muxname);

        return -ENODEV;
}

//该函数通过 mode0. muxmode format 的格式查找相应的 pin mux 属性
//通常传入的参数只有 mode0,mode0 是指相应的 pin 在 mode0 时的名字.
static int __init
omap_mux_get_by_name(const char *muxname,
            struct omap_mux_partition **found_partition,
```

```
            struct omap_mux  ** found_mux)
{
    struct omap_mux_partition  * partition;

    //遍历所有 partition,查找 muxname 的 pin mux,通过传址参数
    //返回相应的 partition 和 omap_mux,函数返回相应的名字
    //和该 pin 的第几个 mode
    list_for_each_entry(partition, &mux_partitions, node) {
        struct omap_mux  * mux = NULL;
        int mux_mode = _omap_mux_get_by_name(partition, muxname, &mux);
        if(mux_mode  <  0)
            continue;

        //找到相应的 pin mux 参数返回 partition 和 omap_mux
        * found_partition = partition;
        * found_mux = mux;

        //返回符合相应 muxmode 的 mode 值 omap3 为 0 ~ 7
        return mux_mode;
    }

    return  - ENODEV;
}

//该函数是其他模块的接口,做相应 pin mux 的配置.
//val 为 input enable, pull up/pull down.
//name 为 mode0. muxmode 模式
int __init omap_mux_init_signal(const char  * muxname, int val)
{
    struct omap_mux_partition  * partition = NULL;
    struct omap_mux  * mux = NULL;
    u16 old_mode;
    int mux_mode;

    //首先通过 muxname 找到相应的配置信息,以及参数使用的 mode
    mux_mode = omap_mux_get_by_name(muxname, &partition, &mux);
    if(mux_mode < 0)
        return mux_mode;

    old_mode = omap_mux_read(partition, mux -> reg_offset);
    //将要设置的 mode 值和要设置的 val 值组合.
    mux_mode  | = val;
```

```
        pr_debug("%s: Setting signal %s 0x%04x  ->0x%04x\n",
                 __func__, muxname, old_mode, mux_mode);
        //写 pin mux 寄存器。
        omap_mux_write(partition, mux_mode, mux->reg_offset);

        return 0;
}

//针对 omap_hwmod,对 pin mux 进行初始化配置,将 hwmod 的所有 pin 进行
//初始化配置,并返回相应的 omap_hwmod_mux_info,供 omap_hwmod 使用。
//由于该函数使用 GFP_KERNEL 分配 memory,所以不能在中断中调用.
//Linux 内核还没有在所有设备模块使用该接口,不过该接口对
//omap_hwmod 使用的引脚设置还是很有意义的
struct omap_hwmod_mux_info * __init
omap_hwmod_mux_init(struct omap_device_pad  *bpads, int nr_pads)
{
        struct omap_hwmod_mux_info  *hmux;
        int i;

        //如果传入参数没有包含 pad 属性则直接返回
        if(! bpads || nr_pads  < 1)
            return NULL;

        //分配相应的 omap_hwmod_mux_info 空间
        hmux = kzalloc(sizeof(struct omap_hwmod_mux_info), GFP_KERNEL);
        if(! hmux)
            goto err1;
        //记录该 module 使用的 pad 数目
        hmux->nr_pads = nr_pads;

        //为相应的 pad 分配 memory, pad 信息包含 pin mux 寄存器信息
        //以及 offmode 时 pin 的属性信息
        hmux->pads = kzalloc(sizeof(struct omap_device_pad)  *
                          nr_pads, GFP_KERNEL);
        if(! hmux->pads)
            goto err2;

        //为 omap_hwmod_mux_info 的每个 pad 信息赋值
        for(i = 0; i  <  hmux->nr_pads; i ++ ) {
            struct omap_mux_partition  * partition;
            struct omap_device_pad  * bpad = &bpads[i], * pad = &hmux->pads[i];
            struct omap_mux  * mux;
```

```
        int mux_mode;

        //获得 pin mux 属性,并赋值相应的 partition 和 omap_mux 属性
        mux_mode = omap_mux_get_by_name(bpad -> name, &partition, &mux);
        if(mux_mode < 0)
            goto err3;
        if(! pad -> partition)
            pad -> partition = partition;
        if(! pad -> mux)
            pad -> mux = mux;

        //为 pad 名赋值
        pad -> name = kzalloc(strlen(bpad -> name) + 1, GFP_KERNEL);
        if(! pad -> name) {
            int j;

            for(j = i - 1; j >= 0; j -- )
                kfree(hmux -> pads[j]. name);
            goto err3;
        }
        strcpy(pad -> name, bpad -> name);

        //flags 和 offmode 的属性赋值
        pad -> flags = bpad -> flags;
        pad -> enable = bpad -> enable;
        pad -> idle = bpad -> idle;
        pad -> off = bpad -> off;
        pr_debug("%s: Initialized %s\n", __func__, pad -> name);
    }

    //返回相应的 omap_hwmod_mux_info 供 omap_hwmod 使用.
    return hmux;

err3:
    kfree(hmux -> pads);
err2:
    kfree(hmux);
err1:
    pr_err("%s: Could not allocate device mux entry\n", __func__);

    return NULL;
```

```
}

/* Assumes the calling function takes care of locking */
//该接口提供给 omap_hwmod 驱动使用,当 omap_hwmod 状态发生变化时
//如 enable, idle,shutdown 如果此时有 pin 设置需求,对相应的 pin 进行设置
//时调用该接口.
//pad 通过其 flag 参数表示 pad 的当前状态.
void omap_hwmod_mux(struct omap_hwmod_mux_info *hmux, u8 state)
{
    int i;

    for(i = 0; i < hmux->nr_pads; i++) {
        struct omap_device_pad *pad = &hmux->pads[i];
        int flags, val = -EINVAL;

        //通过 flag 表示 pad 当前的状态
        //pad 属性包含 OMAP_DEVICE_PAD_REMUX 时在 idle 和 disable
        //状态下的时候才可以配置相应的 pad->idle 和 pad->off 的值
        flags = pad->flags;

        switch(state) {
        case _HWMOD_STATE_ENABLED:
            if(flags & OMAP_DEVICE_PAD_ENABLED)
                break;
            //如果是 enable,则应设置 flag 配置 pad->enable 的值
            flags |= OMAP_DEVICE_PAD_ENABLED;
            val = pad->enable;
            pr_debug("%s: Enabling %s %x\n", __func__,
                    pad->name, val);
            break;
        case _HWMOD_STATE_IDLE:
            if(!(flags & OMAP_DEVICE_PAD_REMUX))
                break;
            //如果是 idle 并且 OMAP_DEVICE_PAD_REMUX
            //则应设置 flag 配置 pad->idle 的值
            flags &= ~OMAP_DEVICE_PAD_ENABLED;
            val = pad->idle;
            pr_debug("%s: Idling %s %x\n", __func__,
                    pad->name, val);
            break;
        case _HWMOD_STATE_DISABLED:
        default:
```

```
                /* Use safe mode unless OMAP_DEVICE_PAD_REMUX */
                //如果 OMAP_DEVICE_PAD_REMUX 则配置 pad->off 的值
                //否则应该设置为 safemode
                if(flags & OMAP_DEVICE_PAD_REMUX)
                    val = pad->off;
                else
                    val = OMAP_MUX_MODE7;
                //设置 flag 值
                flags &= ~OMAP_DEVICE_PAD_ENABLED;
                pr_debug("%s: Disabling %s %x\n", __func__,
                         pad->name, val);
            };

            //写 pin mux 寄存器.
            if(val >= 0) {
                omap_mux_write(pad->partition, val,
                pad->mux->reg_offset);
                pad->flags = flags;
            }
        }
    }
```

从以上分析可见，TI 提供的引脚管理已经包括了引脚总的管理，以及专门针对设备模块的引脚管理，其中还涉及了电源管理相关的功能，是比较完整的架构。只是完全基于 TI 处理器的特点进行设计的，并没有考虑进一步的抽象，没有将操作和属性分别抽象出来，这一点与 pin control 架构还是有差别的，但是从功能完整性的角度来看已经是完整的方案了，可以帮助理解 pin control 架构。在设备模块引脚管理的代码中可见 omap_hwmod，该结构是所有设备模块的抽象，与系统管理以及电源管理息息相关，该结构虽然与 TI 处理器有紧密关系，但也是非常清晰合理的上层抽象，所以作为设备模块的资源之一的引脚管理信息自然应该包含在其中。对于 omap_hwmod 相关的代码会在 TI 的 SoC 电源管理部分进行详细介绍。

4.10 小结

本章主要涉及 Linux 内核核心功能。这些功能是内核重要的横切功能，又属于系统的底层框架，也是与系统移植相关的重要功能。总的来说，这部分是和芯片相关的底层核心功能模块。

内核初始化部分是和系统中各个模块都息息相关的，也是进行板级定制以及系统移植的重要部分。

地址映射是与内核关系紧密的部分，毕竟现代处理器都是使用的虚拟地址，这样在内核层面就需要地址映射，而对设备驱动首先要做的就是设备相关的寄存器映射，可以说地址映射是系统执行的基础，对地址映射的理解体现了对系统层面的整体把握。

中断处理和内存管理都是 Linux 内核的重要组成部分，中断处理主要涉及中断控制器的实现。内存管理大部分的设计都是与体系结构无关的，体系结构相关部分大多可以通过参数设置完成。深入了解内存管理对理解 Linux 内核至关重要，相应的参数设置对于完善系统、提高效率也是十分有意义的。

其他部分如 DMA、时钟管理、时间管理、GPIO 以及引脚复用都为系统提供了操作接口，它们作为公共模块为系统的其他模块提供相应的服务。理解这些服务的框架就可以更好地使用相关的服务实现特定的功能。

第 5 章　内核设备管理以及驱动基础框架

本章所涉及的内容在层次上基本属于桥接层、虚拟层等上层的框架及实现，主要介绍设备管理和驱动相关的功能模块。这些模块为实际的设备管理和驱动提供基础框架和服务，可以帮助驱动程序的开发人员开发体系结构无关的代码，从而使得驱动程序本身更关注于设备的操作而不是关注包含处理器体系结构在内的整个系统。

5.1　VFS 及其与设备的关联

Linux 系统的一个基本思想就是“一切皆是文件”，就是说除了传统意义的文件之外，目录、设备等都是以文件的方式面向用户。这一思想体现了一种高度抽象。这种高度抽象对使用者来说代表了统一的操作界面，有了统一的界面，应用就可以将关注点放在数据处理本身，而不是获得数据的操作方法上。

在 Linux 系统中，实现“一切皆是文件”的模块就是 VFS（Virtual File System）。要实现该功能，需要在框架层进行高级抽象，不仅要考虑文件，还要对文件有一个统一的组织视角，不仅要有各种文件，还要让应用能够按照一定的方式和逻辑找到文件。文件组织的规范是在 FHS（文件系统层次规范）中进行说明，由于属于上层系统的规范，就不进行详述了，但是要明确文件的组织方式是非常重要的。

谈到虚拟文件系统要考虑实际的文件系统，实际的文件系统就是虚拟文件系统的实例化。虚拟文件系统将文件系统管理的实体以及管理的操作进行抽象，而实际的文件系统就将这些抽象实例化，这就要求 VFS 本身还要对实际文件系统的管理实体进行管理。

最后要明确 VFS 的用户是谁？是应用程序也是用户层的进程。这一点明确了，才更容易理解 VFS。

5.1.1　VFS 框架

VFS 的实现是以“一切皆是文件”为需求出发点的。要理解 VFS 的框架首先看一下 VFS 和系统的静态关系框图，如图 5-1 所示。

从图 5-1 可见，VFS 是用户层的直接接口，是面向用户的服务。其封装了不同文件系统的差异，这样用户只把注意力集中在 VFS 中即可。在 VFS 中有 dcache 和 inode cache，这两部分是两种管理实体的 kmem_cache，cache 中分别存放着管理实体 dentry（directory entry）和 inode（index node）。这两种管理实体分别代表文件管理的两个重要方面，一方面是文件组织管理，另一方面是物理文件管理，包括元数据管理（包括时间、权限等）、文件数据管理以及相关的操作管理。

1. 文件组织管理

VFS 中 dentry 代表文件组织的管理实体，能够通过它以树状结构来对各种类型的文件进

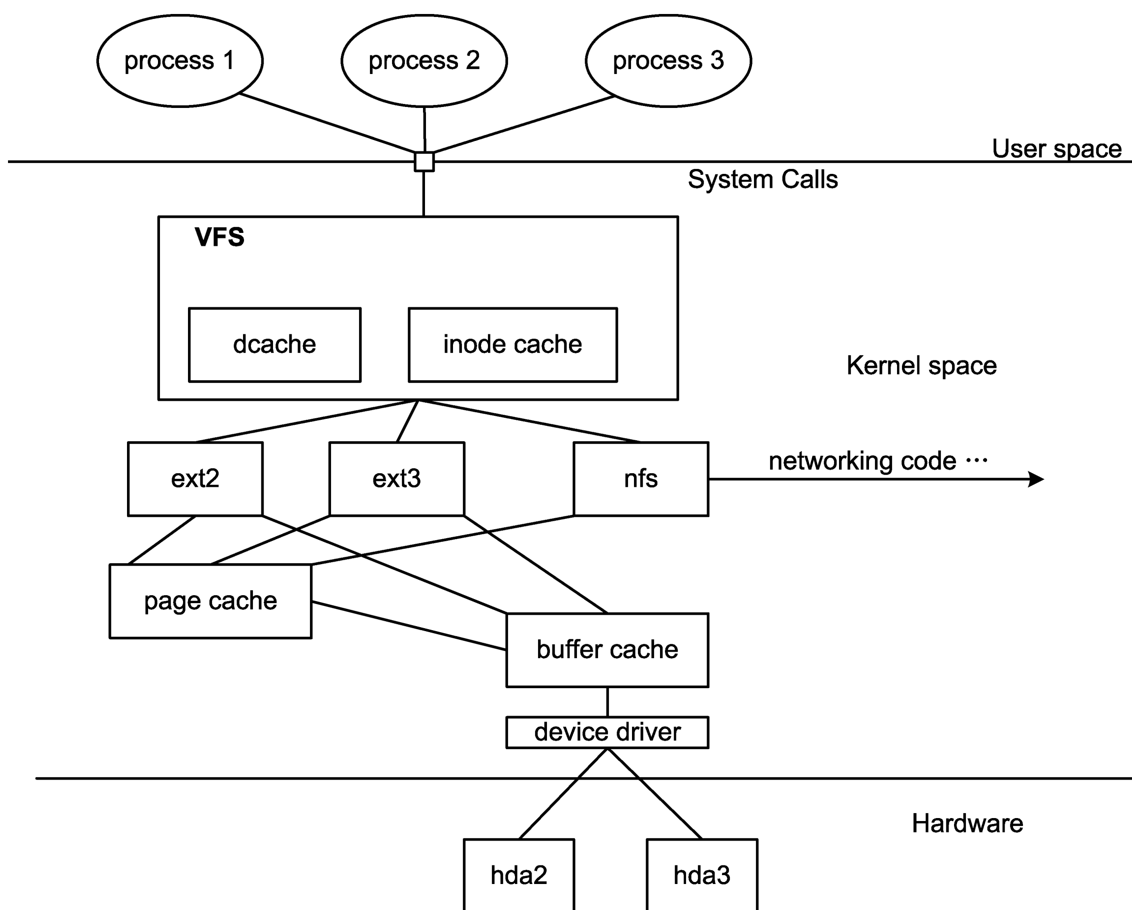


图 5-1　VFS 和系统静态关系图

行管理，其中树状结构的根是通过 d_alloc_root 进行分配的。注意这里说的是树状结构而不是目录结构，树状结构只是表示组织形式，而目录结构容易和目录文件混淆从而产生歧义。dentry 表示系统运行时文件的组织，文件的组织是以文件名及层次的方式实现的，先看看 Linux 系统是如何组织这些文件的（要符合 FHS 标准），如图 5-2 所示。

在图 5-2 中每个节点被使用时都会在 VFS 层中创建 dentry，这样可以快速通过文件名进行查找和定位，Linux 内核中对 dentry 的管理组织形式如图 5-3 所示。

dentry 的 d_subdirs 字段指向子目录项的第一个元素，子目录项之间通过 d_child 相互链接。dentry 的 d_parent 字段指向目录项的父目录项，如果是文件系统的根，则指向自己。这些属性就可以建立文件的树状层次组织管理结构；dentry_hashtable 和 d_hash 则是为了能够加速文件的查找，另外 dentry 作为动态资源也是需要管理的，相应的管理算法由 dentry_unused 组成的链表进行管理，在必要的时候进行资源释放。这样在功能、性能资源各方面都对文件名组织进行了管理。

仅有组织的管理是不够的，还要能通过文件名访问其中的数据。来看看 dentry 数据结构的内容，明确如何从组织转到具体的文件以及其他的管理。

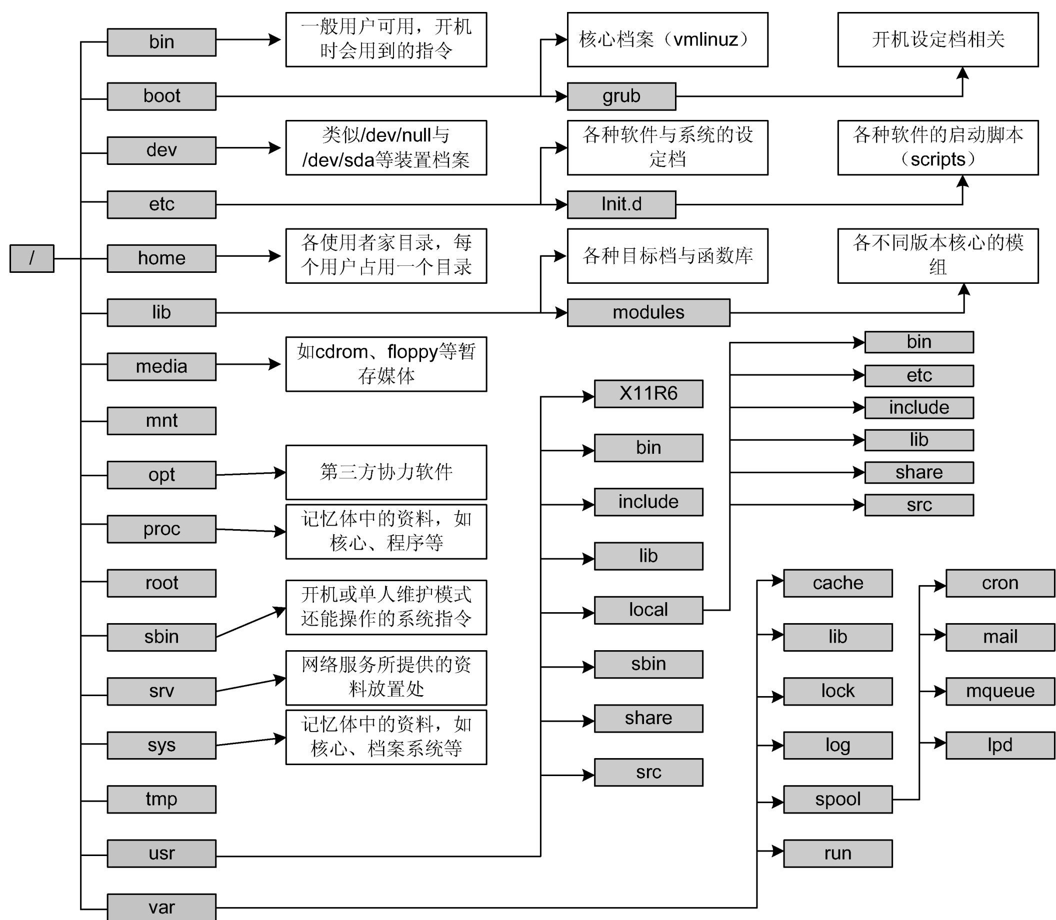


图 5-2　Linux 文件组织层次

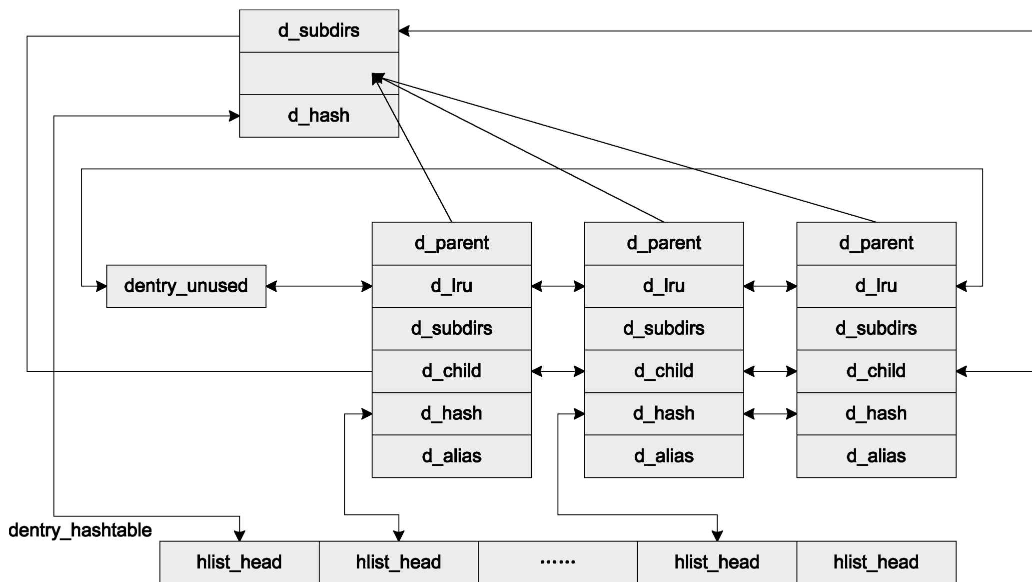


图 5-3　VFS 层 dentry 管理组织形式

```
struct dentry {
    atomic_t d_count;
    unsigned int d_flags;           /* protected by d_lock */
    spinlock_t d_lock;              /* per dentry lock */
    int d_mounted;
    struct inode *d_inode;          /* Where the name belongs to - NULL is
                                     * negative */
    /*
     * The next three fields are touched by __d_lookup.  Place them here
     * so they all fit in a cache line.
     */
    struct hlist_node d_hash;       /* lookup hash list */
    struct dentry *d_parent;        /* parent directory */
    struct qstr d_name;

    struct list_head d_lru;         /* LRU list */
    /*
     * d_child and d_rcu can share memory
     */
    union {
        struct list_head d_child;   /* child of parent list */
        struct rcu_head d_rcu;
    } d_u;
    struct list_head d_subdirs;     /* our children */
    struct list_head d_alias;       /* inode alias list */
    unsigned long d_time;           /* used by d_revalidate */
    const struct dentry_operations *d_op;
    struct super_block *d_sb;       /* The root of the dentry tree */
    void *d_fsdata;                 /* fs-specific data */

    //当文件名较短时使用此区域加速访问
    unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
};
```

在 dentry 结构中除了之前介绍的与组织管理相关的属性外，还有一些比较重要的属性，分别如下：

- d_inode 指向 inode 实体的指针，inode 表示文件的具体属性。
- d_name 表示文件的名字。qstr 是 quick string 的缩写包含字符串、长度和 hash 值，这里用它是对文件名进行封装以便进行快速查找。如果文件名很长，通过文件名计算 hash 值会很费时，这里可以空间换时间。
- d_op 是对文件名的一些计算和操作接口。VFS 针对通常的文件名操作已经提供了基本的接口，但是对特定的文件系统，标准的操作是不能满足需要的。如 fat32 文件系

统对文件名不区分大小写，这时进行文件名比较就需要特定的操作接口。在 VFS 中定义 dentry_operations 相关接口，具体文件系统根据需要对文件名的操作进行重定义来满足文件系统自身的需要。

- d_sb 指向 dentry 对象所在文件系统 super block 的实体。为什么需要该指针呢？因为在 Linux 系统中，对用户来说，文件组织整个是一棵树。但实际情况是在这一棵树上挂载了不同的文件系统，而每个文件系统本身就是一棵树，所以表示文件名的管理实体 dentry 同样应该和所在的文件系统（子树）关联（通过该指针实现）。用户看到的树是多个文件系统子树拟合形成的，拟合的过程涉及挂载点，由 d_mounted 表示是否为挂载点，这样可以在文件名的层次上关联文件系统的挂载。实际挂载的管理将在后续介绍。

以上是系统运行时 VFS 对文件名的组织管理，这是 VFS 管理的一个方面。把文件名和物理文件管理分离，这样在 VFS 层就很容易实现符号文件的快速定位。

2. 物理文件管理

VFS 管理的另一方面是系统运行时对物理文件的管理。相应的管理实体是 inode。注意 inode 是 VFS 层对物理文件的管理实体，主要由于 VFS 是抽象文件系统，其中的管理实体应该是抽象的概念，而 inode 在 VFS 中代表物理文件。这就有些奇怪了，如何做到抽象概念到物理概念的转换呢？了解了其中的过程就明确了如何从 VFS 过渡到实际文件系统，可以说 inode 是抽象文件系统 VFS 到物理文件系统的关键点。在面向对象的设计中通常通过继承关系表示具体实例化，C 语言则是通过包含方式来实现实例化，以 ext2 文件系统为例：

```
struct ext2_inode_info {
    __le32   i_data[15];
    __u32    i_flags;
    __u32    i_faddr;
    __u8     i_frag_no;
    __u8     i_frag_size;
    __u16    i_state;
    __u32    i_file_acl;
    __u32    i_dir_acl;
    __u32    i_dtime;
...
    struct mutex truncate_mutex;
    struct inode vfs_inode;
    struct list_head i_orphan;          /* unlinked but open inodes */
};
```

其中包含 VFS 的 inode 即 vfs_inode，之前提到的 VFS inode cache 实际是各个具体文件系统 inode_info 的 cache，如 ext2 就是 ext2_inode_info 的 kmem_cache。在 VFS 中，对物理文件进行管理的实体在运行中的数据会直接对应到具体文件系统的物理文件管理实体，而 inode 是从各种文件系统物理文件管理中抽象出来的属性集合，注意这些都是运行时的信息。要了解如何从 inode 转换到具体的文件系统操作，首先需要看看 inode 的具体内容：

```
struct inode {
    struct hlist_node              i_hash;
    struct list_head               i_wb_list;      /* backing dev IO list */
    struct list_head               i_lru;          /* inode LRU list */
    struct list_head               i_sb_list;
    struct list_head               i_dentry;
    unsigned long                  i_ino;
    atomic_t                       i_count;
    unsigned int                   i_nlink;
    uid_t                          i_uid;
    gid_t                          i_gid;
    dev_t                          i_rdev;
    unsigned int                   i_blkbits;
    u64i_version;
    loff_t                         i_size;
...
    struct timespec                i_atime;
    struct timespec                i_mtime;
    struct timespec                i_ctime;
    blkcnt_t                       i_blocks;
    unsigned short                 i_bytes;
    umode_t                        i_mode;
    spinlock_t                     i_lock;         /* i_blocks, i_bytes, maybe i_size */
    struct mutex                   i_mutex;
    struct rw_semaphore            i_alloc_sem;
    const struct inode_operations  *i_op;
    const struct file_operations   *i_fop;         /* former ->i_op->default_file_ops */
    struct super_block             *i_sb;
    struct file_lock               *i_flock;
    struct address_space           *i_mapping;
    struct address_space           i_data;
...
    struct list_head               i_devices;
    union {
        struct pipe_inode_info     *i_pipe;
        struct block_device        *i_bdev;
        struct cdev                *i_cdev;
    };

    __u32                          i_generation;
...
    unsigned long                  i_state;
```

```
        unsigned long        dirtied_when;        /* jiffies of first dirtying */

        unsigned int         i_flags;
    ...
        atomic_t             i_writecount;
    ...
        void                 *i_private;          /* fs or device private pointer */
    };
```

从内容可见 inode 包含大量的信息。

首先类似于 dentry，通过链表在 VFS 层中对 inode 完成各种不同的分类管理。对文件的最后访问时间、最后修改时间和最后修改 inode 的时间分别由 i_atime、i_mtime 和 i_ctime 进行管理。文件大小相关信息由 i_size 和 i_blocks 表示。使用两种表现方式是因为文件系统的物理内容通常保存在块设备中，所以分别通过字节大小和所占块数表示文件的大小，以方便进行操作。i_rdev 表示文件所在实际的设备号（如果是设备则是设备号）。另外 inode 中还通过 i_mode 体现了"一切皆是文件"。其中可以表示特殊的文件，也可通过以下的宏表示不同类型的文件。

```
    #define   S_IFSOCK   0140000
    #define   S_IFLNK    0120000
    #define   S_IFREG    0100000
    #define   S_IFBLK    0060000
    #define   S_IFDIR    0040000
    #define   S_IFCHR    0020000
    #define   S_IFIFO    0010000
```

如果 inode 表示设备，还可以通过 union 中的具体设备指针直接关联到设备的结构，i_devices 会和设备管理实体 cdev 或者 bdev 关联。

在关联到具体的文件系统操作方面就要依靠操作接口。inode 中有 i_op 和 i_fop 两种操作接口，前者主要是面向文件的元数据及与 inode 相关的特殊操作，后者则面向文件中数据的操作，通过这些操作接口的重定向就可以转向实际的文件系统的操作，另外由于文件可以 mapping 到虚拟地址空间，也需要转换到具体文件系统给定的操作中，这个转换由 i_mapping 中相关的操作接口完成，VFS 还为 mapping 的方式设计了 direct IO 框架，以便进行相关的操作，这样就与 buffer 以及 page cache 直接关联起来。

这样整个系统就完成了虚拟文件系统和实际文件系统的转换，这几种重要的数据结构已在 VFS 中进行介绍，所以不再对实际的文件系统进行详述，实际文件系统主要是对块设备的空间进行组织和管理以存放实际的数据。

看了这么多，都还是在 VFS 层的管理实体，并没有见到如何从进程角度理解文件并和 VFS 相关实体进行关联，这些都属于系统运行时的状态信息。其关联如图 5-4 所示。

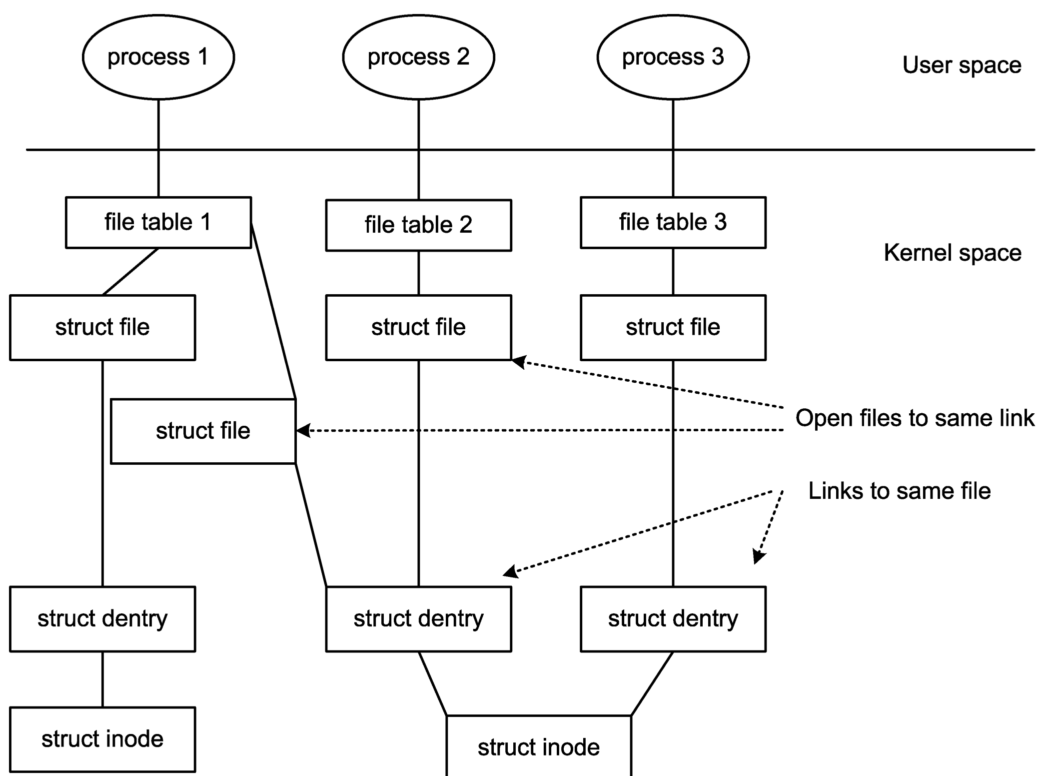


图 5-4　VFS 和进程动态关联图

从图 5-4 可见，在用户进程中文件就是 file 结构体，该结构会在文件打开时创建。下面来看看其主要内容：

```
struct file {
    /*
     * fu_list becomes invalid after file_free is called and queued via
     * fu_rcuhead for RCU freeing
     */
    union {
        struct list_head    fu_list;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path             f_path;
#define f_dentry            f_path.dentry
#define f_vfsmnt            f_path.mnt
    const struct file_operations                *f_op;
    spinlock_t              f_lock;         /* f_ep_links, f_flags, no IRQ */
...
    atomic_long_t           f_count;
    unsigned int            f_flags;
```

```
        fmode_t             f_mode;
        loff_t              f_pos;
        struct fown_struct  f_owner;
        const struct cred   * f_cred;
        struct file_ra_state f_ra;

        u64                 f_version;
    …
        /* needed for tty driver, and maybe others */
        void                * private_data;
    …
    };
```

注意这里通过 path 结构表示路径（path 结构中 dentry 表示所在子树的文件层次属性，mnt 表示整体文件树的层次），path 中的 dentry 可与物理文件的管理实体 inode 关联起来。为了方便文件内容的操作，file 中的 f_op 与 inode 中的 f_op 有关联，但不一定相同，这样可以按需变化，提高系统的灵活性，后续介绍设备的时候再进行详细介绍。

3. VFS 与进程关联

从整体上看看系统是如何将 VFS 内部以及和进程进行关联的，细节如图 5-5 所示。

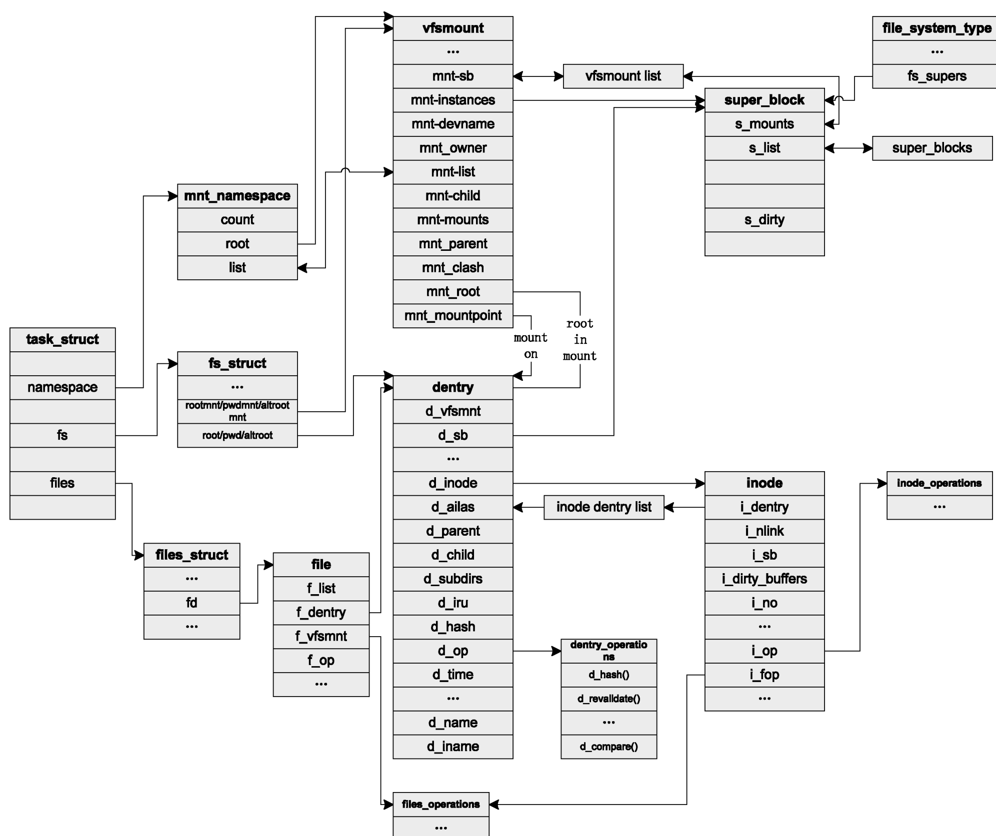


图 5-5　VFS 和进程数据结构关系图

图 5-5 中还有一些数据结构没有说明。mnt_namespace 可以实现用户文件系统的虚拟化管理；vfsmount 则是用来管理文件系统的挂载，之前见到的 path 就是通过 vfsmount 将不同的文件系统子树拟合成用户所见的文件路径。挂载同样是有层次的，相应的层次关系由 vfsmount 的链表以及与文件层次管理 dentry 的关联来完成，具体到属性就是 mnt_parent 表示挂载的层次；mnt_mountpoint 表示挂载的上层文件系统中的目录文件 dentry；mnt_root 表示挂载的文件系统的根目录文件 dentry，另外对于上层的 vfsmount 会通过其链表 mnt_mounts 与下层 vfsmount 的 mnt_child 进行关联，这样文件系统挂载的整个层次就形成了。

以一个例子来理解文件树状组织在内核中是如何体现的。要形成如图 5-6 所示的文件层次。

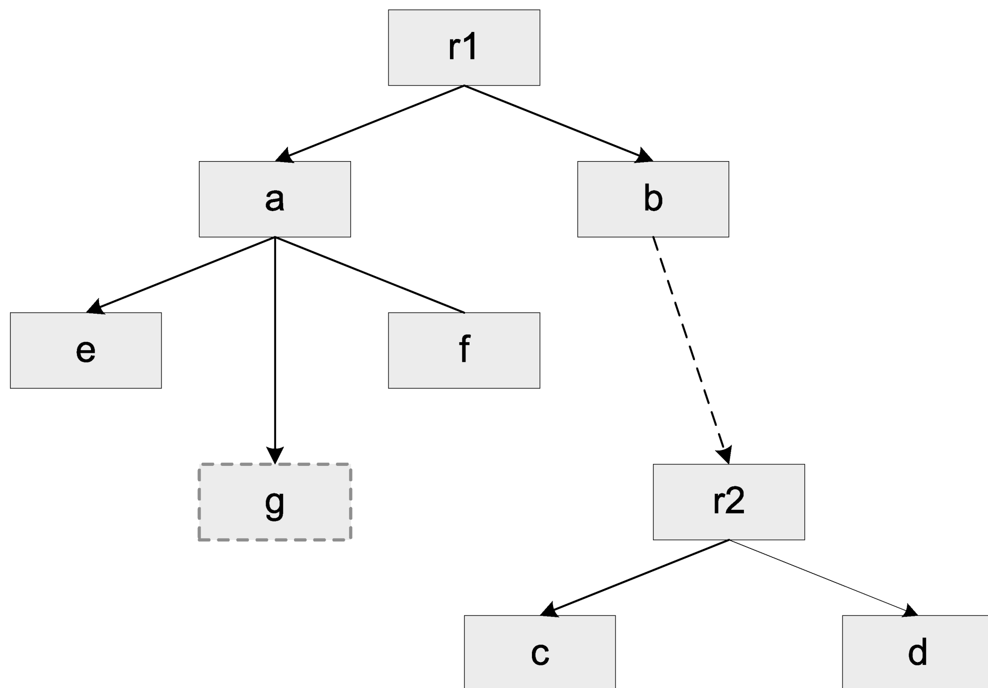


图 5-6　文件层次示例

图 5-6 中可见有两个文件系统，一个作为根挂载到 r1 目录文件，另一个挂载到文件系统的 b 目录文件上，g 文件并没有实际的打开过。对应的 dentry 实际情况如图 5-7 所示。对应的 vfsmount 实际情况如图 5-8 所示。从图 5-7 可见 g 没有被访问所以并没有 g 文件对应的 dentry，这正说明相关内容都是当前系统运行状态的体现。

有了以上的示例，基本可以理解文件层次在内核运行状态下如何表示了。

4. 具体文件系统管理

剩下的还有就是对具体文件系统的管理，在 VFS 中同样要对具体的文件系统进行管理，相应的有 file_system_type 和 super_block，前者管理的是内核中注册的所有文件系统，后者主要是管理文件系统的元数据（包括实际设备以及块设备组织的元信息等）和文件系统通过元数据管理文件的操作接口。这些都与具体的文件系统关系紧密就不进行详述。

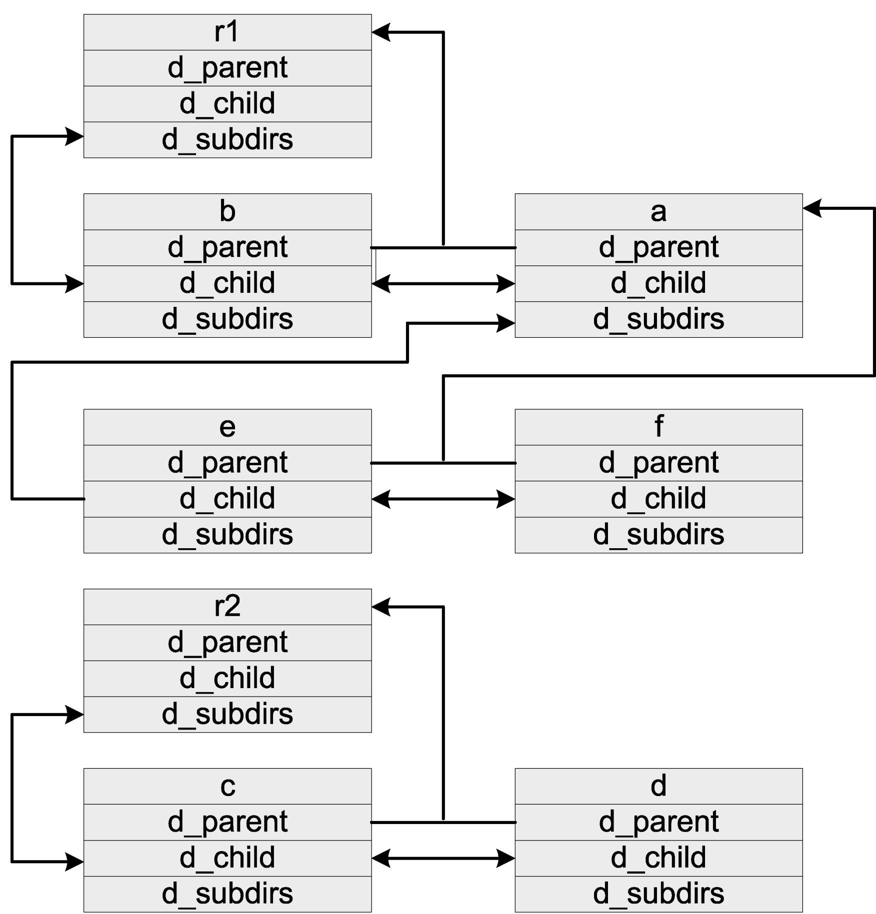


图 5-7　示例文件层次 dentry 实际表示

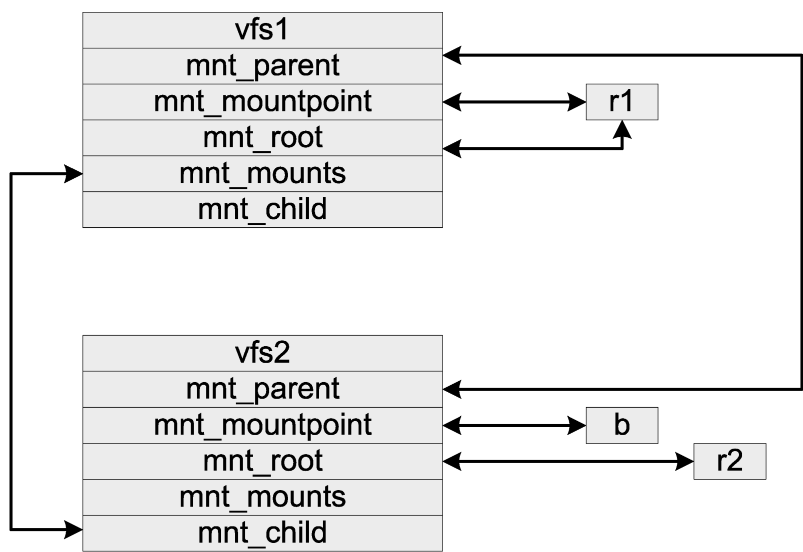


图 5-8　示例文件层次 vfsmount 实际表示

5. VFS 操作的用户通知

文件作为用户的接口，除了具体的操作功能之外，操作本身也是用户关心的一个方面，这就需要内核能够提供对文件进行相关操作的通知，这些通知直接嵌入到 VFS 文件操作接口中即可，相当于监测统计点，这也是 VFS 提供的功能，这些监测统计通过 fsnotify_xxx 上报给事件管理模块 inotify，而 inotify 负责管理用户的监测请求并通知用户，内核提供监测接口如下：

- fsnotify_move 文件从一个目录移动到另一个目录。
- fsnotify_nameremove 文件从目录中删除。
- fsnotify_inoderemove 自删除。
- fsnotify_create 创建新文件。
- fsnotify_mkdir 创建新目录。
- fsnotify_access 文件读。
- fsnotify_modify 文件写。
- fsnotify_open 文件打开。
- fsnotify_close 文件关闭。
- fsnotify_xattr 文件的扩展属性被修改。
- fsnotify_change 文件被修改或原数据被修改。

内核会将如下的事件上报应用层：

```
#define ALL_FSNOTIFY_EVENTS(FS_ACCESS | FS_MODIFY | FS_ATTRIB | \
                FS_CLOSE_WRITE | FS_CLOSE_NOWRITE | FS_OPEN | \
                FS_MOVED_FROM | FS_MOVED_TO | FS_CREATE | \
                FS_DELETE | FS_DELETE_SELF | FS_MOVE_SELF | \
                FS_UNMOUNT | FS_Q_OVERFLOW | FS_IN_IGNORED | \
                FS_OPEN_PERM | FS_ACCESS_PERM | FS_EXCL_UNLINK | \
                FS_ISDIR | FS_IN_ONESHOT | FS_DN_RENAME | \
                FS_DN_MULTISHOT | FS_EVENT_ON_CHILD)
```

应用程序可以通过 inotify 接口进行所关心文件的监测。inotify 属于应用层的内容，就不进行详述了。

至此整体上在 VFS 层为用户提供的各种功能都进行了框架分析和总结。

5.1.2 VFS 与设备关联

由于 VFS 是“一切皆是文件”的实现者，设备作为特殊的文件统一与 VFS 有关联，理解这种关联就可以明确内核如何通过统一的接口来对设备进行操作。

首先，VFS 层通过 init_special_inode 为设备特殊文件提供了操作接口，具体内容如下：

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if(S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
```

```
        } else if(S_ISBLK(mode)) {
            inode->i_fop = &def_blk_fops;
            inode->i_rdev = rdev;
        } else if(S_ISFIFO(mode))
            inode->i_fop = &def_fifo_fops;
        else if(S_ISSOCK(mode))
            inode->i_fop = &bad_sock_fops;
        else
            printk(KERN_DEBUG "init_special_inode: bogus i_mode(%o) for"
                        " inode %s:%lu\n", mode, inode->i_sb->s_id,
                        inode->i_ino);
    }
```

正如之前所述由 inode 中的 i_mode 来进行设备特殊文件的设定。相应的 i_fop 对应的 def_chr_fops、def_blk_fops 分别是字符设备和块设备在设备文件打开时进行操作的接口，属于上层的抽象操作。其中会根据设备号等细节选择合适的设备接口进行操作，细节会在介绍相应设备时详述。实际的设备号则保存在 i_rdev 中。这样 VFS 中文件的管理实体 inode 就和设备关联了。

具体文件系统会有两部分操作与设备文件相关，分别是创建时和打开时。创建时是由具体文件系统的 mknod 进行操作，打开时会获得 inode（需要检查属性是否为设备文件），这两部分都会在需要的时候调用 init_special_inode，这样整个系统就可以将设备统一的作为文件来看待，少量的局部特殊操作带来全局统一的接口，其设计与实现还是十分巧妙的。

5.2 Linux 设备模型（Linux device model）

5.2.1 设备模型的需求及基本设计

设备可以说是在内核管理中最具复杂性和多样性的部分，如图 5-9 所示。从图 5-9 可见，设备类型如此繁杂，需要管理各种不同并且差异极大的设备，在系统层面又要对它们进行统一管理，实现难度非常大。主要的难度体现在既要适应这些多样性还要提供统一的视角来进行管理。

设备管理要对各种不同设备进行生命周期管理、电源管理、设备发现以及驱动动态绑定和管理，这就要求有一个良好的模型对不同的设备在各个层面进行管理。这些是针对设备模型，在内核内部资源管理方面提出的需求。另外一个重要的需求是站在用户的角度，希望设备模型能够提供给用户一个框架，通过该框架用户可以方便地观察设备状态和设置设备属性。

1. 物理设备管理的整体抽象模型

先来看看设备模型如何满足物理设备管理的需求，在介绍硬件的时候可以看到很多物理设备都是挂在物理总线上的，而处理器内部特别是 SoC 内部的设备虽然有片内总线，但是片内总线是硬件逻辑并没有软件抽象的管理实体，这样内部的设备就显得散乱，为了完成以上的管理需求，内核将整个系统的物理设备进行了一定的抽象组织，形成了抽象总线加物理总

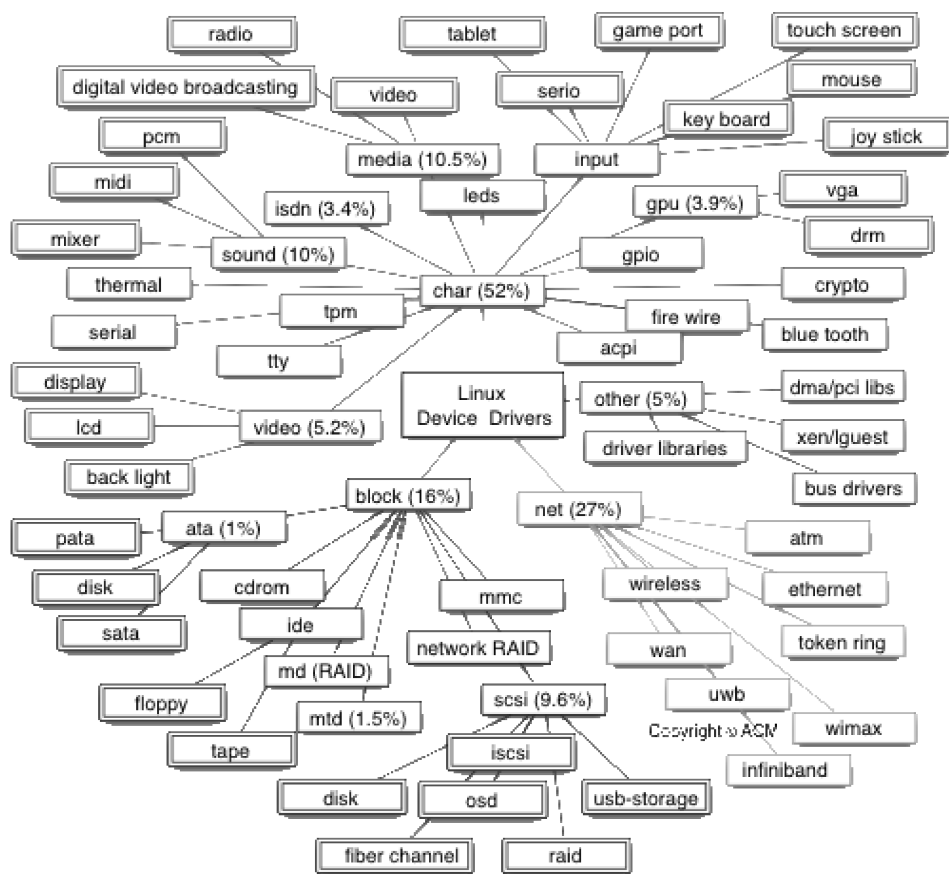


图 5-9　Linux 内核设备复杂多样性

线的框架结构如图 5-10 所示。

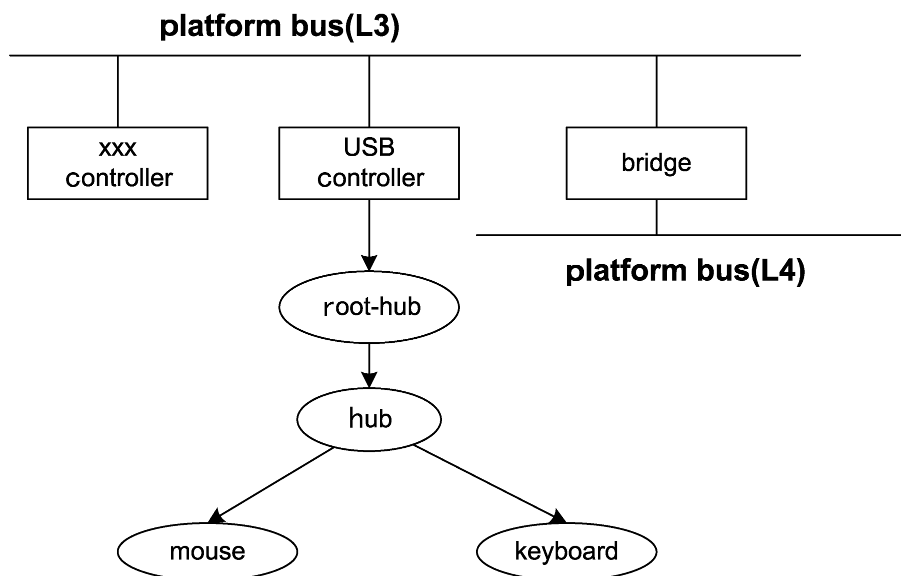


图 5-10　设备模型的抽象组织

从图 5-10 可见，设备模型通过抽象的 platform bus 将与处理器联系紧密（片内设备）但逻辑并不相关的物理设备组织在一起，这样从整体上整个物理系统就形成了总线加设备的组织结构。另外由于设备是需要相应地操作才能工作，需要将这些操作抽象成为设备驱动。对整个物理设备系统进行总结，只需要总线、设备驱动和设备三个管理实体就可以完成管理物理设备的需求，这些总线上的物理设备相应的生命周期、驱动绑定以及管理就可以通过总线来完成。

在介绍设备分类的时候，将设备分为功能型设备和总线型设备，这里看到的模型是通过总线管理各种物理设备，但是此模型并没有完全站在用户使用的视角考虑，用户的视角更关心的是设备的功能，所以同样需要进行功能型设备的管理，虽然在物理上它们是一个设备，但是逻辑上应该分开管理（逻辑功能设备会和物理设备关联），于是将功能型设备的管理放入功能框架中进行就显得比较合理，只是要与设备模型框架进行交互。

2. 观察、设置和管理设备的解决方案

对用户从各种角度方便地观察和设置设备的需求，最好采用 Linux 统一的方式——文件系统，内核针对性的设计了 sys 文件系统框架来满足这些需求。整个 sys 系统框架如图 5-11 所示。

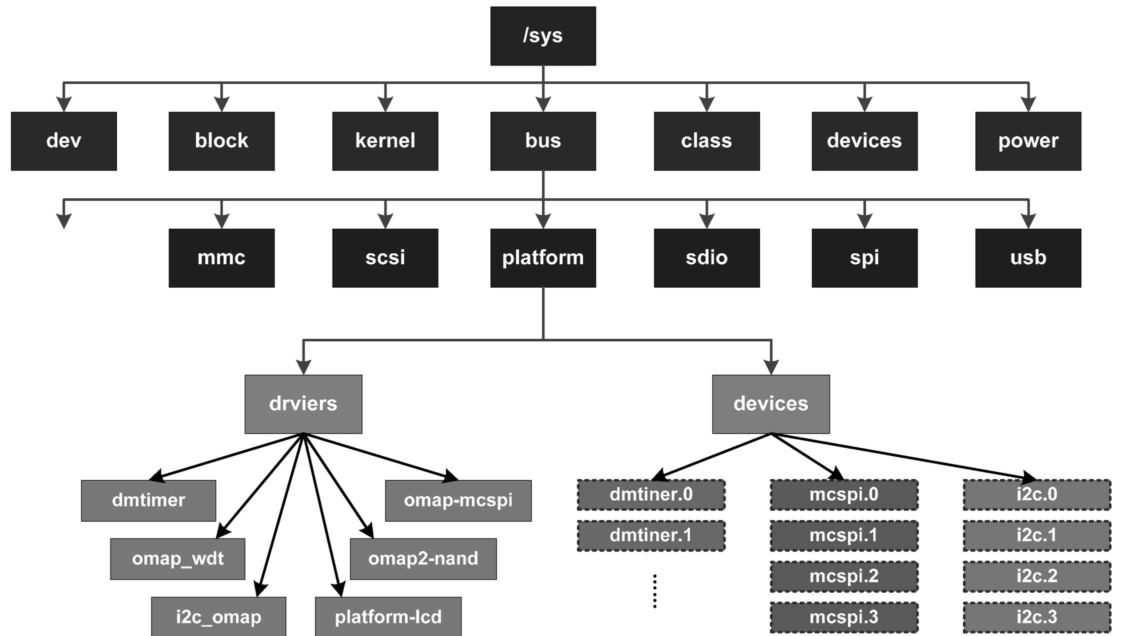


图 5-11　sys 文件系统组织

从图 5-11 可见，sys 文件系统提供了总线（bus）管理的视角和功能分类（class）管理的视角，另外还提供了整体的电源管理以及系统参数的设置。

内核中数据结构与 sys 文件系统关系如图 5-12 所示。

从图 5-12 可见，实际的设备和驱动是通过 kobject 来进行管理的，而上层的 kset 和 subsystem 更多是用来组织相关的底层设备模型中的设备和驱动，而最下层的 attribute（属性）是与设备和驱动等相关的实体表示为 sys 文件系统的文件。

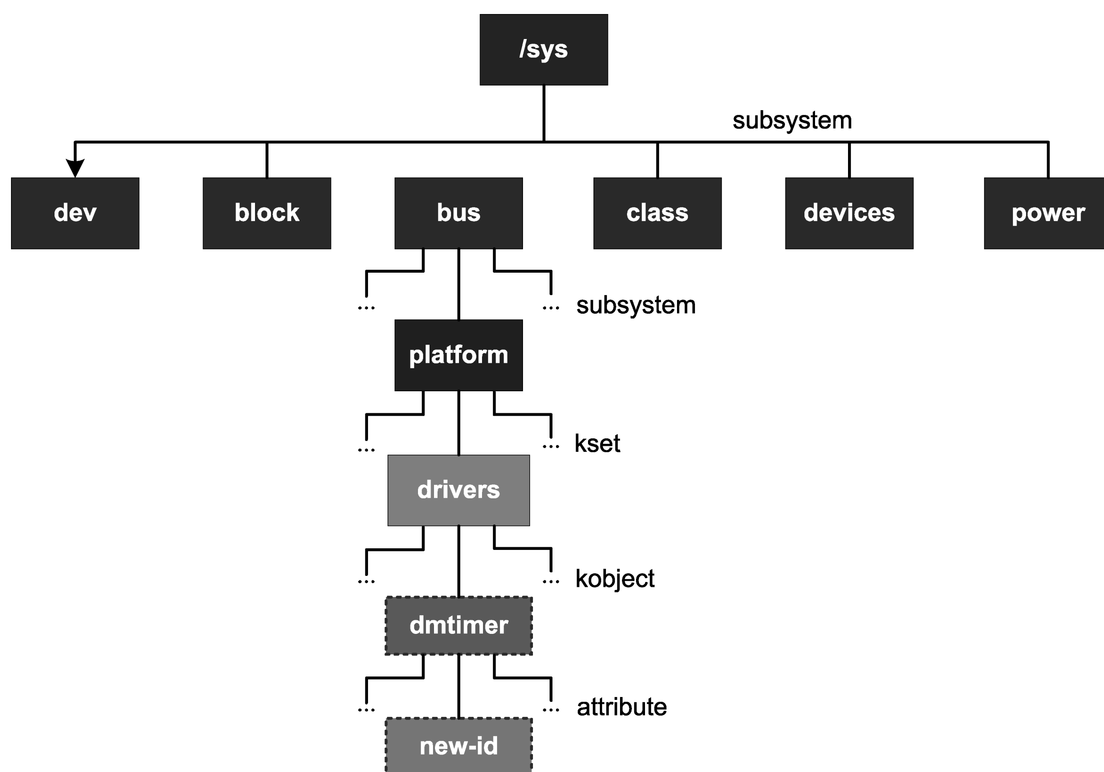


图 5-12　sys 文件系统与数据结构关系

先来看看重要的数据结构 kobject，其内容如下：

```
struct kobject {
    const char                  * name;
    struct list_head            entry;
    struct kobject              * parent;
    struct kset                 * kset;
    struct kobj_type            * ktype;
    struct sysfs_dirent         * sd;
    struct kref                 kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

kobject 承担了两部分的管理功能。其一是通过 kref 承担了生命周期管理（通过引用计数实现）功能，kobject 是高度的抽象实体，对具体的实体（注意这些实体可以不是设备或者驱动，而是一些模块属性）生命周期的状态和管理则通过内嵌 kobject 来完成，这样需要

抽象的 kobject 到具体实体的转换，这种转换通过 ktype 来实现，转换通常是在具体的实体释放以及 sys 文件系统的操作中需要。

针对这些功能，内核提供了一系列接口进行相关操作，首先是引用计数的接口，具体如下：

```
void kref_init(struct kref *kref);
void kref_get(struct kref *kref);
int kref_put(struct kref *kref, void(*release)(struct kref *kref));
```

这些接口主要是针对引用计数的增减进行操作，release 中可以进行抽象到具体的转换，相应的允许内核除 kobject 之外包含 kref 的实体，进行生命周期的管理。

对设备模型中的 kobject 内核提供如下的接口进行生命周期的管理：

```
extern struct kobject *kobject_get(struct kobject *kobj);
extern void kobject_put(struct kobject *kobj);
```

下面以 kobject_put 为例看一下是如何实现的：

```
void kobject_put(struct kobject *kobj)
{
    if(kobj) {
        if(!kobj->state_initialized)
            WARN(1, KERN_WARNING "kobject:'%s' (%p): is not "
                    "initialized, yet kobject_put() is being "
                    "called.\n", kobject_name(kobj), kobj);
        kref_put(&kobj->kref, kobject_release);
    }
}
```

可见其就是使用 kref 的接口进行，其中有相对于 kref 具体的转换接口 kobject_release。kobject_release 中会调用 kobject_cleanup，下面来看看 kobject_cleanup 的具体实现：

```
static void kobject_cleanup(struct kobject *kobj)
{
    //通过 kobj_type 进行具体的实例形态的转换
    struct kobj_type *t = get_ktype(kobj);
    const char *name = kobj->name;

    pr_debug("kobject:'%s' (%p): %s\n",
         kobject_name(kobj), kobj, __func__);

    //没有进行实体的实例化需要报错
    if(t && !t->release)
        pr_debug("kobject:'%s' (%p): does not have a release() "
```

```
                    "function, it is broken and must be fixed. \n",
                    kobject_name(kobj), kobj);

        /* send "remove" if the caller did not do it but sent "add" */
        //如果状态是发送了 add 事件没有发送 remove 事件就发送 remove 事件通知
        if(kobj->state_add_uevent_sent && ! kobj->state_remove_uevent_sent) {
            pr_debug("kobject:'%s' (%p): auto cleanup'remove'event\n",
                 kobject_name(kobj), kobj);
            kobject_uevent(kobj, KOBJ_REMOVE);
        }

        /* remove from sysfs if the caller did not do it */
        //在 sys 文件系统中实体还没有删除则进行删除操作
        if(kobj->state_in_sysfs) {
            pr_debug("kobject:'%s' (%p): auto cleanup kobject_del\n",
                    kobject_name(kobj), kobj);
            kobject_del(kobj);
        }

        //进行实例实体的释放工作
        if(t && t->release) {
            pr_debug("kobject:'%s' (%p): calling ktype release\n",
                    kobject_name(kobj), kobj);
            t->release(kobj);
        }

        /* free name if we allocated it */
        //释放字符串资源
        if(name) {
            pr_debug("kobject:'%s' : free name\n", name);
            kfree(name);
        }
    }
```

这样就实现了生命周期的管理功能。

kobject 的另一个功能就是完成 sys 文件系统的组织功能，通过 parent、kset、sd 等链接完成该功能。相应的组织关系如图 5-13 所示。

关于设备模型的初始化，相关工作并不需要太早执行只要在具体的设备初始化之前执行即可，图 5-14 展示了内核设备模型在整个系统的初始化的位置及流程，这对理解设备模型有着实际的意义。

从图 5-14 可见，具体的设备模型的初始化流程在系统已经基本初始化完毕后执行，这时候 sys 文件系统作为一种特殊的文件系统已经注册并初始化，具体的设备初始化会在 do_

initcalls 以函数表的形式进行，在这之前就是设备模型初始化的好时机，由 driver_init 来完成，其内部会进行总线管理初始化，功能设备管理初始化，platform 总线初始化等。其中会向 sys 文件系统中添加对应的节点。sys 文件系统中设备相关的初始化则在 devices_init 中执行，细节如图 5-15 所示。

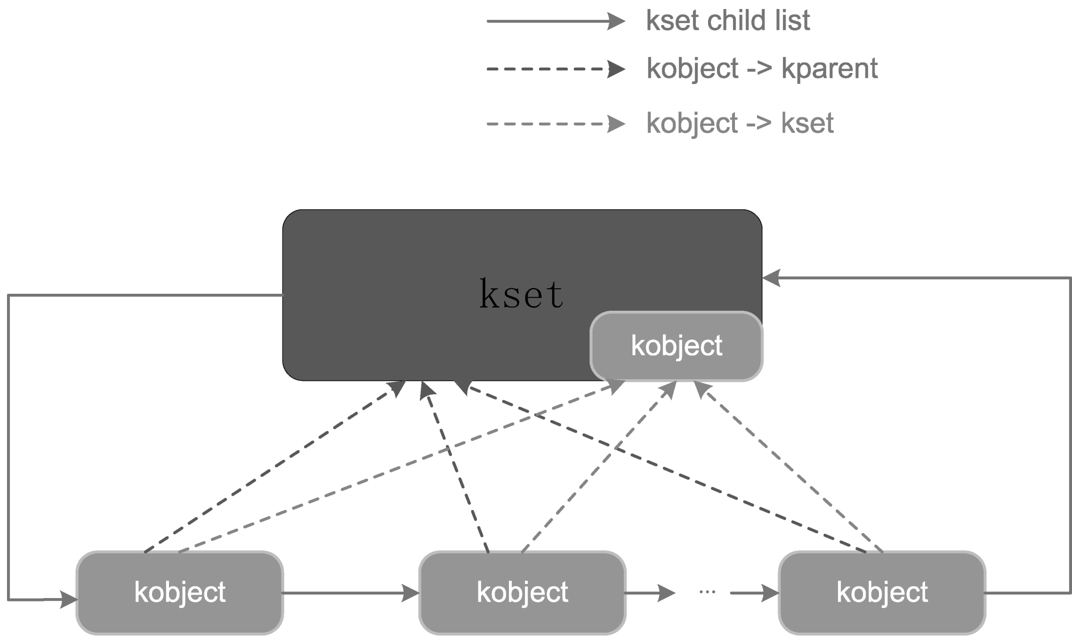


图 5-13　sys 文件系统数据结构组织关系

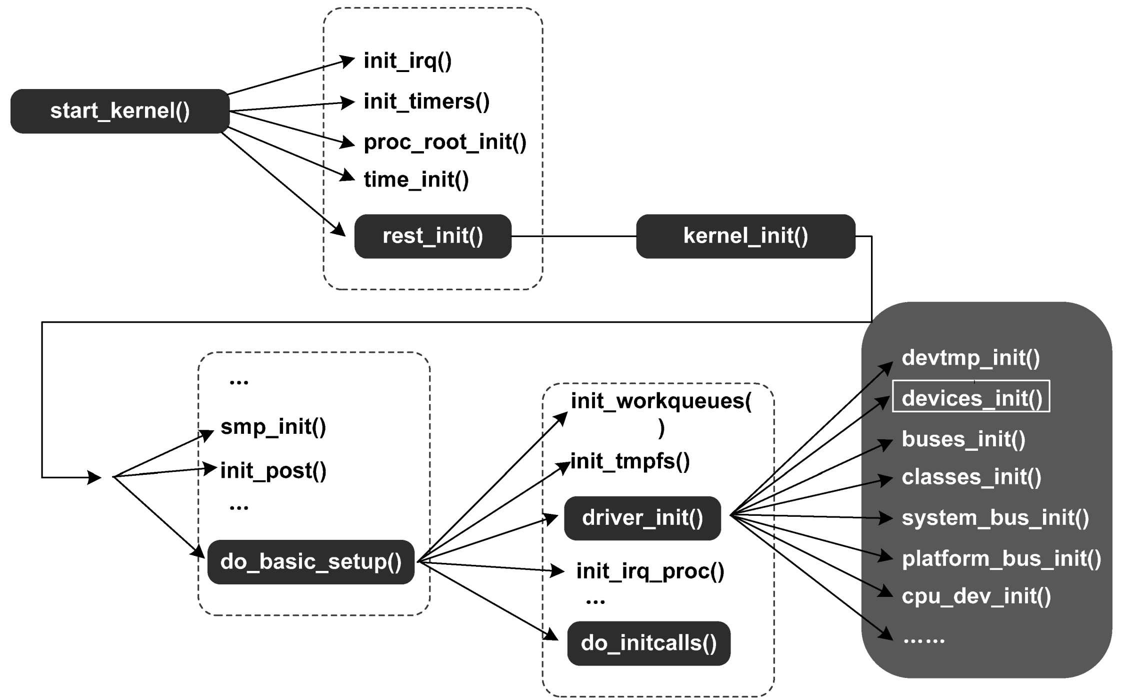


图 5-14　设备模型初始化

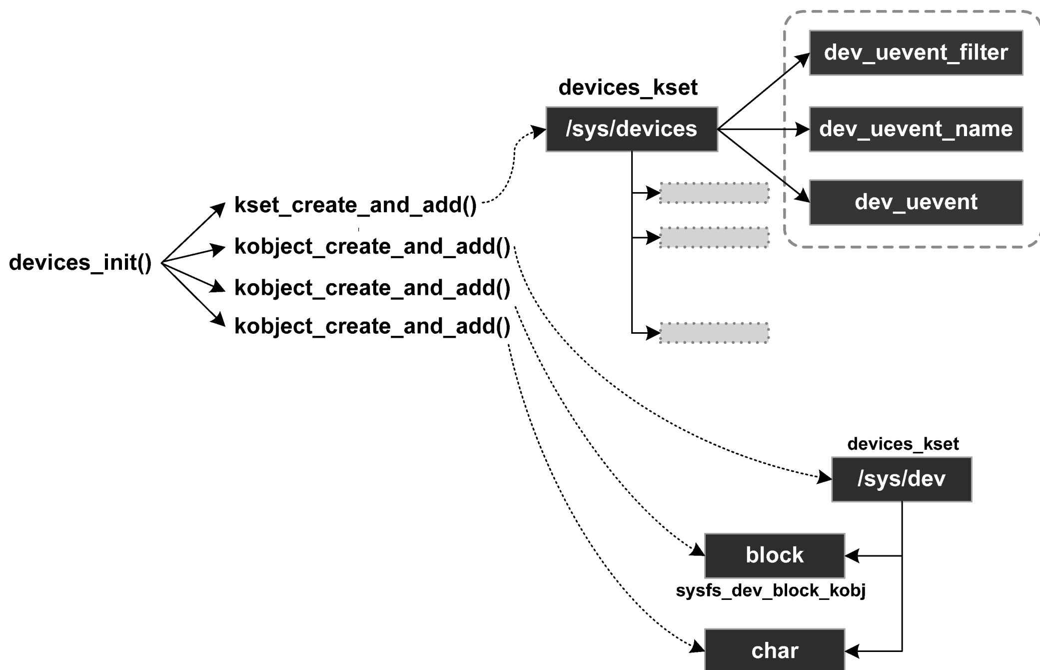


图 5-15　sys 文件系统设备相关初始化

从图 5-15 可见，主要的几个与 device 相关的目录在此创建，为后续实体在相应目录下创建做准备。这样就对设备模型的设计框架有了基本的认识。

5.2.2　总线（bus）

总线是进行设备连接的实体，连接的两端是处理器与设备，处理器需要通过总线来访问和操作设备。在设备模型中总线也承担了连接的功能，由于是在处理器上执行，所以这里连接的是设备和对设备进行操作的实体驱动，其主要功能就是将设备和正确的驱动绑定，从而使用设备，另外针对设备的电源管理提供统一的操作接口。针对这些需求，设备模型抽象出总线的实体 bus_type，具体的内容和分析如下：

```
struct bus_type {
    //总线名称
    const char * name;
    //总线中针对总线,设备和驱动统一的属性
    struct bus_attribute * bus_attrs;
    struct device_attribute * dev_attrs;
    struct driver_attribute * drv_attrs;

    //总线特殊操作用于检查相应的设备和驱动是否匹配
    int( * match)(struct device  * dev, struct device_driver  * drv);
    //总线中有设备添加移除等状态变化时总线的处理函数
    int( * uevent)(struct device  * dev, struct kobj_uevent_env  * env);
```

```
    //这里是强制绑定驱动和设备的接口操作,用于总线中为驱动指定设备时的操作
    int( * probe)(struct device  * dev);
    //设备从总线移除的接口
    int( * remove)(struct device  * dev);
    //总线关闭设备的接口
    void( * shutdown)(struct device  * dev);

    //总线设备需要 sleep 时的接口,这里属于遗留的接口一般都不设置
    int( * suspend)(struct device  * dev, pm_message_t state);
    //设备从 sleep 恢复时总线需要进行操作的接口,这里属于遗留接口一般都不设置
    int( * resume)(struct device  * dev);

    //总线上设备电源管理相关的接口,这里是包含各种电源管理属性的接口
    const struct dev_pm_ops  * pm;

    //这里包含的数据为总线管理的设备和驱动与 sys 文件系统相关的实体
    struct bus_type_private  * p;
};
```

其中总线管理的主要信息都在 bus_type_private，具体信息如下：

```
struct bus_type_private {
    struct kset subsys;
    struct kset  * drivers_kset;
    struct kset  * devices_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type  * bus;
};
```

相应的总线注册接口是 bus_register，其中主要创建 sys 相关节点并填充 bus_type_private 中管理的内容。

下面以 platform bus 为例进行详细分析：

```
struct bus_type platform_bus_type = {
    .name           = "platform",
    .dev_attrs = platform_dev_attrs,
    .match          = platform_match,
    .uevent         = platform_uevent,
    .pm             = &platform_dev_pm_ops,
};
```

其中主要的是 match 接口，用于匹配设备和驱动，细节如下：

```
static int platform_match(struct device  * dev, struct device_driver  * drv)
{
    //转换为总线实际特定的设备和驱动
    struct platform_device  * pdev = to_platform_device(dev);
    struct platform_driver  * pdrv = to_platform_driver(drv);

    /*  Attempt an OF style match first  */
    //首先按照 device tree 的方式检查是否匹配
    if(of_driver_match_device(dev, drv))
        return 1;

    /*  Then try to match against the id table  */
    //如果驱动对管理的设备号有限制需要检查
    if(pdrv -> id_table)
        return platform_match_id(pdrv -> id_table, pdev) != NULL;

    /*  fall - back to driver name match  */
    //通常采用检查名字的方式进行匹配,只要设备和驱动名字相同就是匹配了.
    return(strcmp(pdev -> name, drv -> name) == 0);
}
```

这里检查到匹配后会由相应的驱动通过 probe 接口来调查设备管理需要的资源，并对相应的资源进行申请绑定以及初始化操作，从而最终完成设备和驱动的绑定。

驱动模型中总线相关的初始化细节如图 5-16 所示。

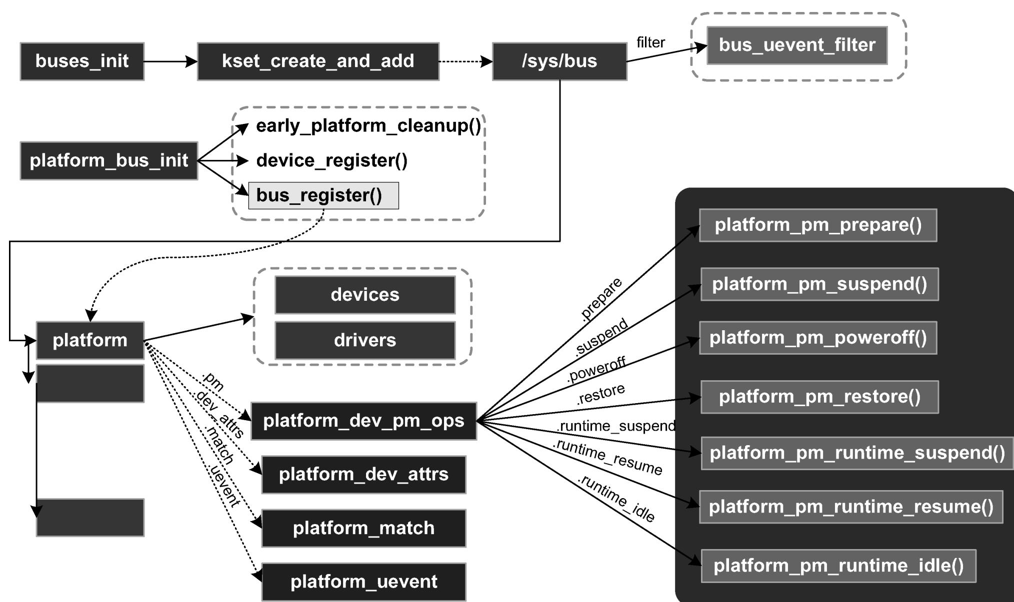


图 5-16　设备模型总线相关初始化

从图 5-16 可见，整体的总线初始化只是创建 sys 文件系统中 bus 节点，而 platform 总线初始化会创建总线节点及管理设备和驱动的节点，这样方便了管理，也为后续加入的设备和驱动提供良好直观的接口。

5.2.3 驱动（driver）

设备模型中驱动的主要功能是和设备绑定后为使用设备提供相关的操作服务。既然是服务就可以为多个设备提供，只要与设备匹配，服务功能就能正确地完成。设备模型中管理驱动的数据结构是 device_driver，其中包含驱动提供的通用服务的接口，下面对它进行分析：

```
struct device_driver {
    //驱动名字
    const char              *name;
    //所属总线
    struct bus_type         *bus;

    struct module           *owner;
    const char              *mod_name;      /* used for built-in modules */

    //取消绑定功能的属性值
    bool suppress_bind_attrs;               /* disables bind/unbind via sysfs */

#if defined(CONFIG_OF)
    //open firmware device tree 相关
    const struct of_device_id     *of_match_table;
#endif

    //匹配到设备后,探测设备资源并申请初始化的接口
    int(*probe)(struct device *dev);
    //设备 remove 时的接口
    int(*remove)(struct device *dev);
    //设备 shutdown 时的接口
    void(*shutdown)(struct device *dev);
    //与 bus 相同遗留接口
    int(*suspend)(struct device *dev, pm_message_t state);
    int(*resume)(struct device *dev);
    const struct attribute_group **groups;

    //电源管理接口
    const struct dev_pm_ops *pm;
```

```
    //在 sys 文件系统中管理的资源
    struct driver_private *p;
};
```

来看看 sys 文件系统中驱动之下主要管理哪些实体：

```
struct driver_private {
    struct kobject kobj;
    struct klist klist_devices;
    struct klist_node knode_bus;
    struct module_kobject *mkobj;
    struct device_driver *driver;
};
```

从 driver_private 可见，其中主要的信息是运行中使用该驱动的设备列表。

device_driver 还是抽象层面的实体，而到具体的总线时，相应的驱动要有与总线相关的驱动实体，因为只有具体总线的驱动才会了解相应总线的设备，相应的服务接口也要与具体的总线设备相关。从这个层面上讲设备模型中的驱动与总线关系更紧密。

以 platform 驱动实体为例进行分析：

```
struct platform_driver {
    int(*probe)(struct platform_device *);
    int(*remove)(struct platform_device *);
    void(*shutdown)(struct platform_device *);
    int(*suspend)(struct platform_device *, pm_message_t state);
    int(*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};
```

其中定义了总线设备相关的服务接口：

```
int platform_driver_register(struct platform_driver *drv)
{
    drv->driver.bus = &platform_bus_type;
    if(drv->probe)
        drv->driver.probe = platform_drv_probe;
    if(drv->remove)
        drv->driver.remove = platform_drv_remove;
    if(drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;
```

```
        return driver_register(&drv->driver);
    }
```

从 platform 驱动注册接口可以看出，抽象层的服务接口都被换为实例化的接口，这样内嵌的 driver 可以通过设备模型核心提供的标准操作进行管理，也可以通过这些实例化接口进行转换。

系统中设备与驱动绑定的流程如图 5-17 所示。

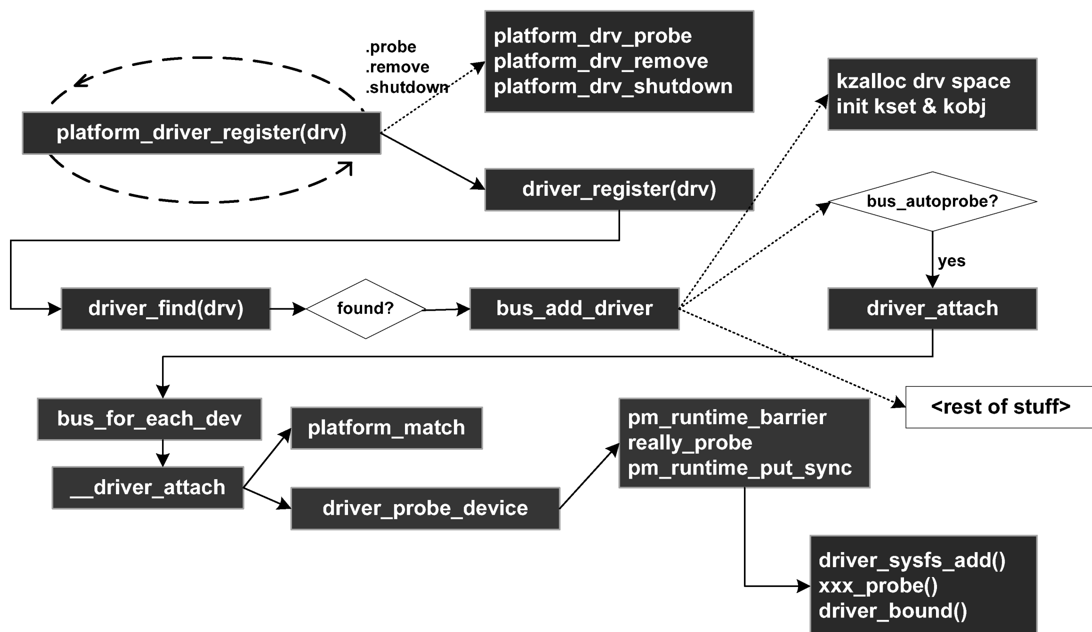


图 5-17　驱动与设备绑定的流程

从图 5-17 可见，具体总线驱动在注册的时候就会根据总线是否自动 probe 设备（通常自动有效）来进行设备的绑定，为了实现该功能设备模型核心提供了很多接口方便设备和驱动的绑定，相应的功能也是十分完善的。其中添加 driver 的接口就是 driver_register，来看看具体的内容：

```
    int driver_register(struct device_driver *drv)
    {
        int ret;
        struct device_driver *other;
        BUG_ON(!drv->bus->p);

        //首先检查接口是否齐全,相应的接口必须有
        if((drv->bus->probe && drv->probe) ||
            (drv->bus->remove && drv->remove) ||
            (drv->bus->shutdown && drv->shutdown))
```

```
                printk(KERN_WARNING "Driver' %s' needs updating - please use "
                        "bus_type methods\n", drv -> name);

    //检查 driver 是否已经注册
    other = driver_find(drv -> name, drv -> bus);
    if(other) {
        printk(KERN_ERR "Error: Driver' %s' is already registered, "
                "aborting...\n", drv -> name);
        return -EBUSY;
    }

    //通过总线添加,这里之前要说明总线类型
    ret = bus_add_driver(drv);
    if(ret)
        return ret;
    //将 driver 设置接口即属性
    ret = driver_add_groups(drv, drv -> groups);
    if(ret)
        bus_remove_driver(drv);
    return ret;
}
```

从代码中可见，设备模型管理的 device_driver 必须与 bus 关联，因为只有 bus 才能帮其找到所管理的设备，而具体的总线都会如 platform 总线一样封装出注册 driver 的接口函数，并将 device_driver 的 bus 属性填入，为上层提供统一的接口，也避免跨层设置。后续的操作如图 5-17 所示，根据 bus 的自动 probe 属性来进行具体总线的设备与驱动的绑定。

5.2.4 设备（devices）

对设备模型中的设备，考虑的重点是如何进行抽象。任何设备要使用都是需要资源的，无论是自身的资源还是系统的资源，所以应从其拥有的资源考虑进行抽象。

具体分析设备模型中的设备管理，首先要分析 device 结构。

```
struct device {
    //表示设备的层次关系
    struct device           *parent;
    //设备模型核心使用,其中有设备实例化私有数据
    struct device_private   *p;

    //生命周期管理实体
    struct kobject kobj;
```

```
    const char          * init_name;            /* initial name of the device */
    //功能类设备和总线类设备中都可能有不同的类型,这里是相应的类型实例化
    struct device_type * type;

    struct mutex        mutex;                  /* mutex to synchronize calls to its driver. */

    //总线类设备所在总线
    struct bus_type     * bus;                  /* type of bus device is on */
    //绑定的驱动
    struct device_driver        * driver;       /* which driver has allocated this device */
    void                * platform_data;        /* Platform specific data, device core doesn't touch
it */
    //设备电源管理需要的信息
    struct dev_pm_infopower;
…
    //设备使用 DMA 时访问资源的限制说明
    u64                 * dma_mask;             /* dma mask(if dma'able device) */
    u64                 coherent_dma_mask;      /* Like dma_mask, but for alloc_coherent mappings as
                                                   not all hardware supports 64 bit addresses for con-
                                                   sistent
                                                   allocations such descriptors. */

    //以下是 DMA 参数及资源属性说明
    struct device_dma_parameters * dma_parms;

    struct list_headdma_pools;/* dma pools(if dma'ble) */

    struct dma_coherent_mem * dma_mem; /* internal for coherent mem override */
    /* arch specific additions */
    struct dev_archdataarchdata;
#ifdef CONFIG_OF
    struct device_node * of_node;
#endif

    dev_t               devt;                   /* dev_t, creates the sysfs "dev" */

    //device resource 的管理实体
    spinlock_t          devres_lock;
    struct list_head    devres_head;

    struct klist_node   knode_class;
    //功能类设备使用,关联到具体的类型
```

```
        struct class             * class;
        const struct attribute_group  ** groups;/ *  optional groups  * /
        //实例化释放的接口
        void( * release)(struct device  * dev);
    };
```

从 device 可见，重点是管理的资源，当然也包含针对 sys 文件系统关联的属性。而设备的层次关系在实际的情况下通常是从逻辑层的功能设备逐渐到物理层的总线设备，最终到 platform bus 中对应的 device，这样系统就建立了完整的设备层次关系。

还是以 platform 总线的设备进行实例化分析：

```
    struct platform_device {
        const char                          *  name;
        int                                 id;
        struct device                       dev;
        u32                                 num_resources;
        struct resource *  resource;

        const struct platform_device_id      * id_entry;

        / *  arch specific additions  * /
        struct pdev_archdata                archdata;
    };
```

这里可以理解为其记录总线相关的特殊资源，但实际又有 platform 总线的特殊性，resources 实际是各种资源的抽象，其中包含中断、IO 空间等。其资源具体的类型在 ioport. h 中定义，具体如下：

```
    #define IORESOURCE_IO          0x 00000100
    #define IORESOURCE_MEM         0x 00000200
    #define IORESOURCE_IRQ         0x 00000400
    #define IORESOURCE_DMA         0x 00000800
    #define IORESOURCE_BUS         0x 00001000
```

可见 platform 设备管理的资源就是物理设备的各种资源，这和实际也是一致的。

整体来说，设备模型中的设备就是管理资源以供驱动访问和操作，资源管理是设备管理的一个核心。

当然设备管理还需要能为用户创建合适的操作接口文件，不能因为创建时间的差异而产生不同的文件名，这就需要和用户层的管理程序结合，相应的框架和流程如图 5-18 所示。

从图 5-18 可见，内核设备模型和 udev 紧密结合，按从 1 ~6 的顺序执行最终对固定的设备产生固定的操作接口文件，从而保证了应用的一致视角。设备模型通知 udev 的方式如图 5-19所示。

通过 uevent 通知到应用层，就完成设备管理创建设备文件到应用层的操作。这样设备

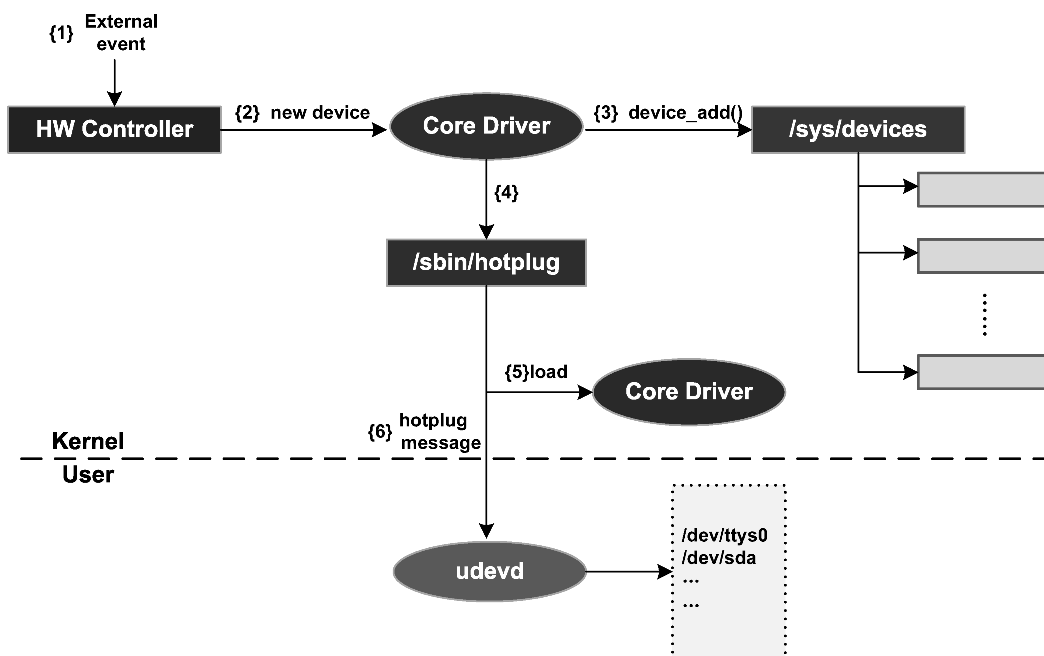


图 5-18　设备模型设备文件创建流程及框架

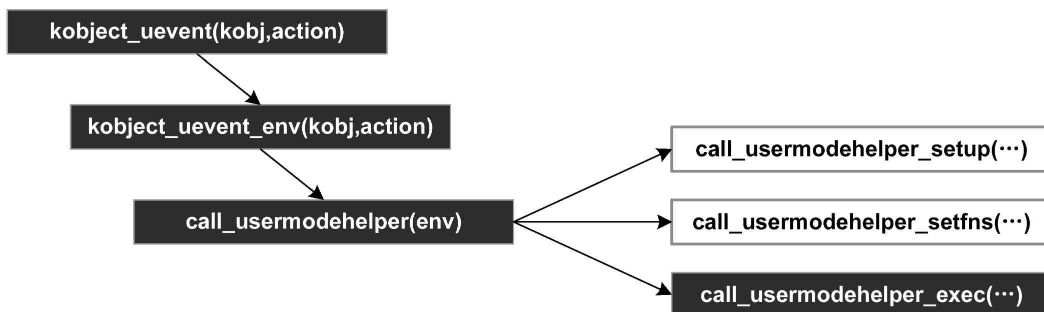


图 5-19　设备模型通知应用层流程

管理的功能就是完整的了。

关于设备名，设备模型提供产生文件名的接口 device_get_devnode，并通过 uevent 属性文件提供给用户层，device_get_devnode 的细节如下：

```
const char *device_get _devnode(struct device *dev,
                    mode_t *mode, const char **tmp)
{
    char *s;

    *tmp = NULL;

    /* the device type may provide a specific name */
    //根据设备类型加入特殊的设备信息,通常用于总线表示不同类型设备
```

```
    if(dev->type && dev->type->devnode)
        *tmp = dev->type->devnode(dev, mode);
    if(*tmp)
        return *tmp;

    /* the class may provide a specific name */
    //加入功能类信息,通常用于具体功能的设备放入同一个目录
    if(dev->class && dev->class->devnode)
        *tmp = dev->class->devnode(dev, mode);
    if(*tmp)
        return *tmp;

    /* return name without allocation, tmp == NULL */
    if(strchr(dev_name(dev),'!') == NULL)
        return dev_name(dev);

    /* replace '!' in the name with '/' */
    *tmp = kstrdup(dev_name(dev), GFP_KERNEL);
    if(!*tmp)
        return NULL;
    while((s = strchr(*tmp,'!')))
        s[0] = '/';
    return *tmp;
}
```

设备模型通过 device_add 接口来为以上功能提供统一的服务，具体细节如下：

```
int device_add(struct device *dev)
{
    struct device *parent = NULL;
    struct kobject *kobj;
    struct class_interface *class_intf;
    int error = -EINVAL;

    //避免 device 被释放
    dev = get_device(dev);
    if(!dev)
        goto done;

    if(!dev->p) {
        error = device_private_init(dev);
        if(error)
```

```
                goto done;
        }
    ...
        //建立设备之间的层次关系
        parent = get_device(dev -> parent);
        kobj = get_device_parent(dev, parent);
        if(kobj)
            dev -> kobj.parent = kobj;

        /* use parent numa_node */
        if(parent)
            set_dev_node(dev, dev_to_node(parent));

        /* first, register with generic layer. */
        /* we require the name to be set before, and pass NULL */
        error = kobject_add(&dev -> kobj, dev -> kobj.parent, NULL);
        if(error)
            goto Error;

        /* notify platform of device entry */
        //芯片特殊的操作,用于芯片内设备加载的特殊操作,通常为空
        if(platform_notify)
            platform_notify(dev);

        //以下创建与应用层设备文件创建逻辑(如 udev)相关的接口,用于读取
        //创建设备文件需要的设备信息
        error = device_create_file(dev, &uevent_attr);
        if(error)
            goto attrError;

        if(MAJOR(dev -> devt)) {
            error = device_create_file(dev, &devt_attr);
            if(error)
                goto ueventattrError;

            //设备号相关信息
            error = device_create_sys_dev_entry(dev);
            if(error)
                goto devtattrError;

            devtmpfs_create_node(dev);
```

```
}

//这里检查设备是否属于具体功能类。如果是,则加入相应 class 中
error = device_add_class_symlinks(dev);
if(error)
    goto SymlinkError;
error = device_add_attrs(dev);
if(error)
    goto AttrsError;
//这里检查设备是否属于具体总线。如果是,则加入相应 bus 中
error = bus_add_device(dev);
if(error)
    goto BusError;
//增加设备电源管理相关的管理属性
error = dpm_sysfs_add(dev);
if(error)
    goto DPMError;
//将设备加入到系统的 pm 管理系统中,主要是设备功耗管理列表
device_pm_add(dev);

/* Notify clients of device addition.   This call must come
 * after dpm_sysfs_add() and before kobject_uevent().
 */
//如果是总线设备,则通知总线有设备加入,关心此事件的模块可以是非总线模块。
if(dev->bus)
    blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                 BUS_NOTIFY_ADD_DEVICE, dev);

//发布新设备的事件,以便 udev 进行后续处理
kobject_uevent(&dev->kobj, KOBJ_ADD);
//如果设备是总线类型设备,这里会根据总线是否自动 probe 来进行驱动绑定
bus_probe_device(dev);
if(parent)
    klist_add_tail(&dev->p->knode_parent,
                   &parent->p->klist_children);

//功能类型设备进行的操作
if(dev->class) {
    mutex_lock(&dev->class->p->mutex);
    /* tie the class to the device */
    //将其加入设备列表
    klist_add_tail(&dev->knode_class,
```

```
                        &dev -> class -> p -> class_devices);

                /* notify any interfaces that the device is here */
                //通知注册的 interface 有设备加入,可以做一些特殊的操作.
                //通常类型并没有注册 interface
                list_for_each_entry(class_intf,
                            &dev -> class -> p -> class_interfaces, node)
                if(class_intf -> add_dev)
                    class_intf -> add_dev(dev, class_intf);
                mutex_unlock(&dev -> class -> p -> mutex);
            }
        done:
            put_device(dev);
            return error;
        …
        }
```

从代码细节可见，设备并不完全与总线绑定，还有上层功能抽象的功能类，这和之前的设备分类是吻合的。具体的设备在进行 add 操作之前要将其 bus 或者 class 的属性进行正确地设置。设备模型提供了良好的框架，具体的实例化就交由具体的模块实现。

5.2.5 功能类（class）

前面已经介绍了功能类实际是站在用户的角度的高级别抽象，用户更关心功能而不是物理的连接，相应的设备模型中也为其提供管理实体。class 结构是主要的管理实体，其内容如下：

```
struct class {
    //类名。
    const char              * name;
    struct module           * owner;

    //同一类功能设备的统一属性
    struct class_attribute  * class_attrs;
    struct device_attribute * dev_attrs;
    //这里是上层抽象的类型实体,实际中是 sys 系统的字符型设备或者块类型设备
    //内核通过该属性将设备加入到/sys/dev 合适的设备下
    struct kobject          * dev_kobj;

    //同一类功能设备的操作接口
    int( * dev_uevent)(struct device * dev, struct kobj_uevent_env * env);
    char * ( * devnode)(struct device * dev, mode_t * mode);
```

```
        void( * class_release)(struct class  * class);
        void( * dev_release)(struct device  * dev);

        int( * suspend)(struct device  * dev, pm_message_t state);
        int( * resume)(struct device  * dev);

        const struct kobj_ns_type_operations  * ns_type;
        const void  * ( * namespace)(struct device  * dev);

        //电源管理接口
        const struct dev_pm_ops  * pm;

        //所管理的信息
        struct class_private  * p;
    };
```

看看 class 所管理的内容：

```
    struct class_private {
        struct kset class_subsys;
        struct klist class_devices;
        struct list_head class_interfaces;
        struct kset class_dirs;
        struct mutex class_mutex;
        struct class  * class;
    };
```

从 class 管理的信息来看只有设备，这是为什么？为什么没有驱动？仔细考虑一下，这些都是从用户的角度考虑的类型，用户对设备的操作接口是文件，所以相应的驱动是通过文件操作接口实现相关功能，这里不对文件进行管理，所以没有驱动实体。

另系统提供了注册 class 的接口，具体如下：

```
    extern int __must_check __class_register(struct class  * class, struct lock_class_key  * key);
```

功能型驱动框架最终都会调用该接口注册功能类。

5.2.6　设备资源管理（device resource）

对设备所需资源进行管理时，驱动经常要处理各种资源分配和释放时的异常情况，相关的代码有随意性、稳定性的问题，内核的开发者针对这种情况开发了统一的接口方便驱动的开发者进行资源的申请和释放，这样可以简化驱动的开发减少冗余代码，使开发者更专注于核心的内容而不是资源的分配和释放。熟悉这些接口对于驱动的开发大有好处，下面列出主要的资源和接口：

内存

```
devm_kzalloc()
devm_kfree()
```

IO 空间

```
devm_request_region()
devm_request_mem_region()
devm_release_region()
devm_release_mem_region()
```

中断

```
devm_request_irq()
devm_free_irq()
```

DMA 资源

```
dmam_alloc_coherent()
dmam_free_coherent()
dmam_alloc_noncoherent()
dmam_free_noncoherent()
dmam_declare_coherent_memory()
dmam_pool_create()
dmam_pool_destroy()
```

IO 映射

```
devm_ioremap()
devm_ioremap_nocache()
devm_iounmap()
```

时钟

```
devm_clk_get()
devm_clk_put()
```

以上基本涵盖了设备驱动开发使用的主要资源，其实现是与设备模型的 device 相结合，这样资源和设备就进行强关联，从而可以简化管理。

5.3 字符设备（char device）

5.3.1 字符设备的特点和需求

字符设备是 Linux 内核管理设备中的一大类设备，在设备模型介绍中可以看到系统初始化的时候在 sys 文件系统下创建了两大类设备，分别是字符设备和块设备，这样做是系统管理的需要，在设备管理中同样要进行分层化的管理，只有这样才能简化 VFS 层的处理，从

而更好地贯彻内核“一切皆是文件”的思想。字符设备和块设备就是层次化设备管理的顶层分类。在进行设备分类时是要以数据为核心，不同类型的设备对数据的组织方式以及数据操作特点是不同的，在顶层进行区分就要在高层抽象考虑。字符设备处理的数据属于流式（stream），特点是数据需要进行顺序处理，不能进行回读。流式数据本身的组织和结构化特点是偏向应用，并不由内核其他模块进行再封装，在层次上就可以直接面向应用层。

应用层直接使用的用户界面（UI）相关的设备都具有以上的特点，所以都归于字符设备，这也是设备层次管理的需要，否则由于增加了不同类型的文件，VFS 和具体文件系统都需要增加复杂度。字符设备框架需要能够区分不同的设备类型并管理相关设备及其操作，另外需要能够为 VFS 层提供设备重载的功能，将正确的设备操作提供给应用层。由于是上层抽象的设备，需要支持各种类型的设备，在上层抽象进行生命周期的管理会便于底层框架的开发，另外需要支持 hotplug（动态加载移除）的设备，这就要求字符设备框架层支持相应的用户设备文件的动态有效性。

5.3.2 字符设备的核心数据结构及操作

下面介绍字符设备的框架，字符设备的核心数据结构是 cdev，具体内容如下：

```
struct cdev {
    //与设备模型关联,负责相关生命周期管理
    struct kobject kobj;
    struct module *owner;
    //设备特定的文件操作接口
    const struct file_operations *ops;
    //与 VFS 中 inode 关联负责文件动态有效性
    struct list_head list;
    //设备号
    dev_t dev;
    unsigned int count;
};
```

从相应的属性中可见，以上的需求都可以通过该结构进行管理。

1. 生命周期管理

首先，在字符设备框架中，生命周期管理是通过 kobj 进行的，相应的管理已经通过字符设备框架进行封装，并通过对外的接口统一提供相关功能。包含生命周期管理的字符设备框架对外接口如下：

```
void cdev_init(struct cdev *, const struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);
void cdev_del(struct cdev *);
```

cdev_get 设计成内部使用接口，在文件打开时通过 chrdev_open 进行调用，这样可以保证引用计数的增加，从而保证不被释放。

下面来看看在 cdev 生命周期结束时的操作：

```
static struct kobj_type ktype_cdev_default = {
        . release        = cdev_default_release,
};

static struct kobj_type ktype_cdev_dynamic = {
        . release        = cdev_dynamic_release,
};

static void cdev_default_release( struct kobject  * kobj)
{
        struct cdev  * p = container_of( kobj, struct cdev, kobj);
        cdev_purge( p);
}

static void cdev_dynamic_release( struct kobject  * kobj)
{
        struct cdev  * p = container_of( kobj, struct cdev, kobj);
        cdev_purge( p);
        kfree( p);
}
```

为什么会有两个很接近的操作呢？这是由于对 cdev 的创建分别有静态地创建 cdev_init 和动态地创建 cdev_alloc 两种，对应用生命周期结束同样有两种、静态使用 cdev_default_release、动态使用 cdev_dynamic_release。动态比静态增加了释放内存的操作。但是 cdev_purge 的工作是什么呢？在需求讨论的时候提到了字符设备是对用户开放使用的，这里设备的生命周期结束时一般是驱动释放操作，相应的 VFS 中代表实际文件 inode 的属性就应该调整，该函数就是将 inode 与 cdev 的关联关闭，后续的打开设备文件操作会重新加载设备驱动，这样才能保证系统的正确性。

2. 设备分类管理

对设备的分类和具体设备的管理，系统通过设备号来进行，设备号是具体设备的类型和实体的综合，与设备一一对应，由于通过设备号进行设备管理是通用的方式，同样适合于块设备，而不同的设备对设备号的管理方式会有差别，所以各自进行单独的设备号管理。

对抽象设备实体如 cdev 的管理，从设计的角度和设备号分离灵活性更高考虑，将二者进行分离管理。

下面详细分析设备号管理，对字符设备的设备号进行管理的核心实体是 char_device_struct，详细内容如下：

```
static struct char_device_struct {
        struct char_device_struct  * next;
        unsigned int major;
        unsigned int baseminor;
        int minorct;
```

```
        char name[64];
        struct cdev *cdev;                /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

字符设备分类比较特殊，并不是只通过主设备号来进行分类的，允许将主设备管理的子设备号分开表示不同类型的设备，作为主类型下面的子类型，每种类型有自己的驱动，char_device_struct 就是用来进行这种分类管理以及和驱动绑定的。可见其中通过 major 主设备号，basemonor 基本的子设备号以及 minorct 子设备数目来表示所管理的设备。相应管理的底层函数是在__register_chrdev_region 中，相应的接口如下：

```
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                         int minorct, const char *name)
```

如果 major 设置为 0，则进行主设备号的动态分配。相应的系统也提供静态申请设备号和动态申请设备号两个接口，分别如下：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

其中 register_chrdev_region 为静态申请主设备号，alloc_chrdev_region 为动态分配主设备号。

3. cdev 实体管理

对字符设备来说，由于其主要为用户直接使用，而没有与内核中除 VFS 外的其他模块关联，所以设计时将 cdev 既作为对上层 VFS 的接口也作为对下层驱动框架的接口，来进行统一管理。另外在内核中针对字符设备和块设备类型设计了统一的设备驱动管理实体的管理框架，主要的数据结构如下：

```
struct kobj_map {
        struct probe {
                struct probe *next;
                dev_t dev;
                unsigned long range;
                struct module *owner;
                kobj_probe_t *get;
                int(*lock)(dev_t, void *);
                void *data;
        } *probes[255];
        struct mutex *lock;
};
```

其中 dev 为驱动管理的起始设备号，range 为管理的范围，data 对字符设备驱动来说就

是指向 cdev（也作为设备驱动管理实体）。

相应的内核提供添加和查找的功能，接口分别如下：

```
int kobj_map(struct kobj_map *domain, dev_t dev, unsigned long range,
             struct module *module, kobj_probe_t *probe,
             int(*lock)(dev_t, void *), void *data)

struct kobject *kobj_lookup(struct kobj_map *domain, dev_t dev, int *index)
```

具体的由 kobj_map 负责添加，由 kobj_lookup 负责查找。

字符设备为了底层设备框架开发方便将添加接口封装为 cdev_ add 以便调用，细节如下：

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
    p->dev = dev;
    p->count = count;
    return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}
```

从代码中可见，其直接调用 kobj_map 来实现功能。

4. 字符设备驱动注册流程

对字符设备驱动来说，由于其管理的设备号和管理实体 cdev 是分开管理的，所以字符设备驱动的注册流程是：首先进行所管理的设备号的申请（可以静态申请也可以动态申请）；然后是管理实体 cdev 的申请（可以静态初始化和动态申请）；最后通过 cdev_add 加入进行统一管理即可。新的字符设备驱动都是采用这种方式注册。

对于字符设备驱动的注册，老的内核使用 register_chrdev 进行，系统需要兼容老的内核，所以提供了接口函数__register_chrdev，而 register_chrdev 就是简单地调用__register_chrdev，下面通过__register_chrdev 来了解新的注册过程。

```
int __register_chrdev(unsigned int major, unsigned int baseminor,
             unsigned int count, const char *name,
             const struct file_operations *fops)
{
    struct char_device_struct *cd;
    struct cdev *cdev;
    int err = -ENOMEM;

    //获得管理的设备号范围
    cd = __register_chrdev_region(major, baseminor, count, name);
    if(IS_ERR(cd))
        return PTR_ERR(cd);
```

```
    //动态分配 cdev 驱动管理实体
    cdev = cdev_alloc( ) ;
    if( ! cdev)
        goto out2;

    //正确的设置参数,主要是文件操作接口
    cdev -> owner = fops -> owner;
    cdev -> ops = fops;
    kobject_set_name( &cdev -> kobj, "%s", name) ;

    //将驱动管理实体加入系统进行管理
    err = cdev_add( cdev, MKDEV( cd -> major, baseminor) , count) ;
    if( err)
        goto out;

    cd -> cdev = cdev;

    return major ? 0 : cd -> major;
out:
    kobject_put( &cdev -> kobj) ;
out2:
    kfree( __unregister_chrdev_region( cd -> major, baseminor, count) ) ;
    return err;
}

static inline int register _chrdev( unsigned int major, const char * name,
                        const struct file_operations * fops)
{
    //老的接口对于子设备只有 256 个的限制.
    return __register_chrdev( major, 0, 256, name, fops) ;
}
```

驱动注册之后，就等待应用程序打开并使用相应的设备了。使用设备的操作都是由各种类型的驱动框架提供，相应的会调用驱动具体的接口函数。

5. 设备操作的重载

字符设备作为上层的抽象设备，要担负起操作重载的工作，这部分工作是在 VFS 中完成的，接下来看看具体的实现：

```
//之前 VFS 中介绍的获得 inode 时加载的操作接口,目的是等到打开设备才进行重载
const struct file_operations def_chr_fops = {
    . open = chrdev_open,
    . llseek = noop_llseek,
```

```
};

static int chrdev_open(struct inode *inode, struct file *filp)
{
    struct cdev *p;
    struct cdev *new = NULL;
    int ret = 0;

    spin_lock(&cdev_lock);
    //这里获得和文件管理相关的 cdev 实体,cdev 作为 VFS 的接口实体
    p = inode->i_cdev;
    //p 为空说明之前没有打开过
    if(!p) {
        struct kobject *kobj;
        int idx;
        spin_unlock(&cdev_lock);
        //根据设备号返回 cdev 中的 kobj。
        kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
        if(!kobj)
            return -ENXIO;
        //获得 cdev 实体
        new = container_of(kobj, struct cdev, kobj);
        spin_lock(&cdev_lock);
        /* Check i_cdev again in case somebody beat us to it while
           we dropped the lock. */
        //由于之前的操作 unlock 了锁,所以有可能其他模块已经进行了设置
        //这里进行一下检查
        p = inode->i_cdev;
        //p 为空说明没有设置过
        if(!p) {
            //将文件和 cdev 关联
            inode->i_cdev = p = new;
            list_add(&inode->i_devices, &p->list);
            //获得 cdev,这里的设置避免 put 释放操作
            new = NULL;
        } else if(!cdev_get(p))
            ret = -ENXIO;
        } else if(!cdev_get(p))
            ret = -ENXIO;
        spin_unlock(&cdev_lock);
        //之前 kobj_lookup 会进行 get 操作,这里已经用完,应该进行 put 操作.
        cdev_put(new);
```

```
        if(ret)
            return ret;

        ret = -ENXIO;
        //这里进行文件操作的重载,使用的是特定驱动框架的接口
        filp->f_op = fops_get(p->ops);
        if(! filp->f_op)
            goto out_cdev_put;

        //进行重载后的打开操作
        if(filp->f_op->open) {
            ret = filp->f_op->open(inode,filp);
            if(ret)
                goto out_cdev_put;
        }

        return 0;

    out_cdev_put:
        cdev_put(p);
        return ret;
}
```

重载对用户来说是不可见的，用户并没有感受到相应的重载过程。这样实现也是满足设备层次管理的需要。

5.3.3 字符设备子类型

字符设备分类可以在系统启动后从 proc 文件系统和 sys 文件系统中获得相关信息。

首先看看/proc/devices 下面的信息：

```
Character devices:
1  mem
4  /dev/vc/0
4  tty
4  ttyS
5  /dev/tty
5  /dev/console
5  /dev/ptmx
5  ttyprintk
6  lp
7  vcs
```

```
 10   misc
 13   input
 14   sound
 21   sg
 29   fb
 81   video4linux
 99   ppdev
108   ppp
116   alsa
128   ptm
136   pts
180   usb
189   usb_device
216   rfcomm
226   drm
251   hidraw
252   bsg
253   watchdog
254   rtc
```

这里只是列出了主设备号，可以看出字符设备中有些设备类型主设备号是重复的。更细节的信息可以从/sys/dev/char 下获得，下面是其中的一部分信息：

```
root root 4:60  ->../../devices/virtual/tty/tty60
root root 4:61  ->../../devices/virtual/tty/tty61
root root 4:62  ->../../devices/virtual/tty/tty62
root root 4:63  ->../../devices/virtual/tty/tty63
root root 4:64  ->../../devices/platform/serial8250/tty/ttyS0
root root 4:65  ->../../devices/platform/serial8250/tty/ttyS1
root root 4:66  ->../../devices/platform/serial8250/tty/ttyS2
root root 4:67  ->../../devices/platform/serial8250/tty/ttyS3
root root 4:68  ->../../devices/platform/serial8250/tty/ttyS4
```

其中包含主设备号、次设备号和路径等信息，从中可以了解具体的设备分类和设备号的划分。详细的设备分类信息可以通过 Documentation/devices. txt 文件获得。后面会对部分设备进行详细的分析。

5.4 块设备（block device）

5.4.1 块设备特点和需求

块设备的数据特点是有固定的数据单元（块），可以随机存取。块设备是非易失存储设

备，其上的数据是可以回读的，只要数据没有改变，任何时间读取同一位置的数据意义都是相同的。

非易失存储设备很重要的功能是存放用户和系统的数据，而用户和系统的数据一般依赖于文件系统，通过块设备对非易失存储设备进行操作，具体的数据结构化和组织留给文件系统进行管理。块设备就是文件系统的物理基础。

从系统的角度，块设备作为系统数据的存放设备，相应的性能是十分重要的。由于可以随机访问，块设备的管理就要比字符设备复杂的多。基于这些原因在块设备框架的设计过程中效率是必须要考虑的因素，字符设备更多考虑的是延时问题，而块设备则是要考虑整个系统的性能。为了提高整体性能，比较普遍的方法是在内存中建立 buffer 通过缓冲来提高性能，这样虽然提高了系统的复杂度，但是为了性能也是值得的。另外有了缓冲之后写操作的实时性要求降低，而读操作在性能方面同样有延时要求。针对整体性能的提升，系统有对数据进行批处理操作的需求。

5.4.2 块设备核心数据结构及操作

1. 整体框架

要了解块设备首先要从系统层面了解块设备所在的位置和上下层的关系，如图 5-20 所示。

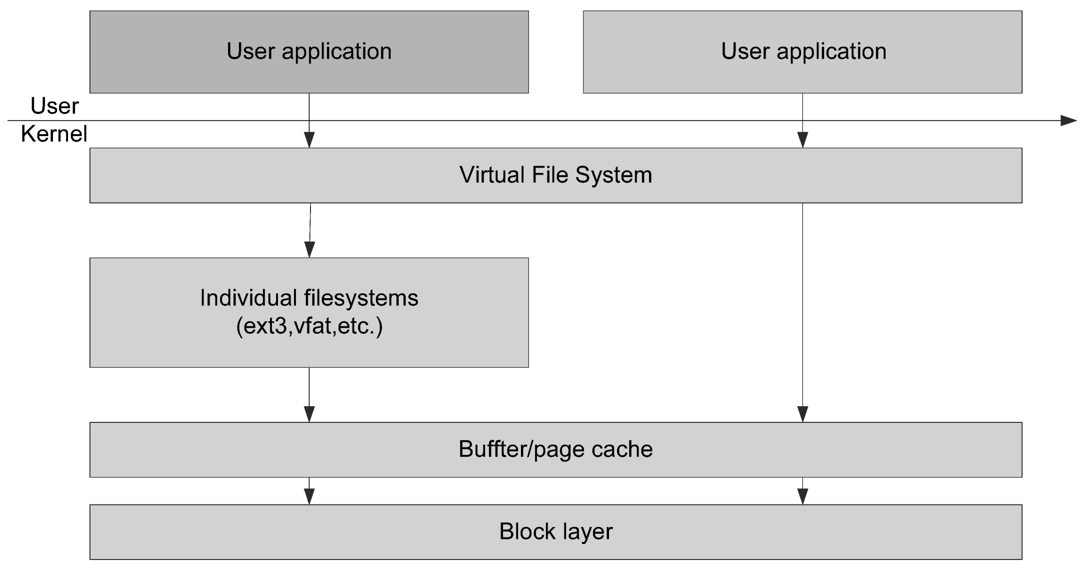


图 5-20 块设备在系统中层次关系

从图 5-20 可见，块设备层既可以作为裸设备来供应用层使用，也可以作为物理文件系统的承载设备与文件系统进行关联。为了提升整个系统的性能，在块设备层之上增加了 buffer 层来缓冲数据，buffer 是缓存在内存中，所以物理上会将多个 buffer 组织在 page 中，整体形成 page cache。这种架构满足了之前涉及的需求，并兼顾灵活性与整体的性能。

块设备层的内部设计同样要考虑性能，从整体上块设备的内部框架如图 5-21 所示。从图 5-21 可见，块设备的框架层次还是很灵活的，可以为了满足某种需求，设计虚拟块设备

实现如负载均衡等高级的功能；为了整体性能和吞吐量，增加了 IO scheduler 层来优化物理设备的读写性能；也可以跳过 IO scheduler 层直接进行块设备操作。总体上讲，这种内部框架可以把分层的驱动综合起来实现复杂的功能。

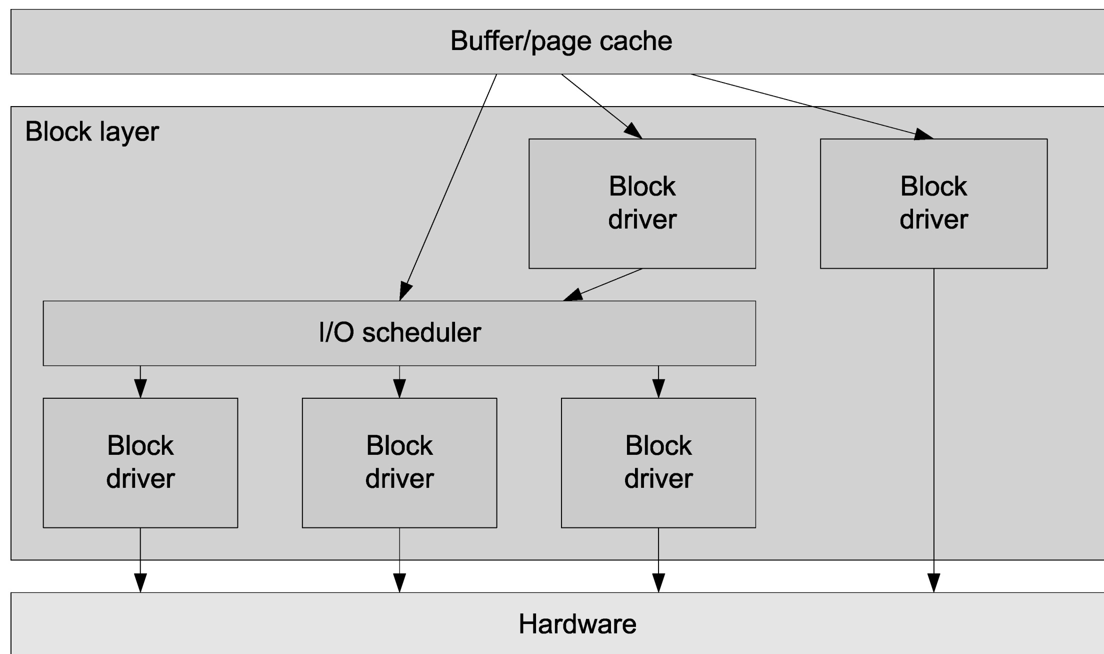


图 5-21　块设备内部框架

2. 核心管理实体

从整体的层次上看，块设备需要有针对文件系统和 VFS 的接口实体，另外还需要管理生命周期的设备模型相关实体以及驱动管理实体。除此之外块设备的数据是以块为单位，自然在操作上要有相关的单位操作管理实体以及管理方法，这些就形成了驱动。

针对以上的分析，来看看 Linux 内核的具体实现，块设备主要管理实体及系统关系如图 5-22所示。

图 5-22 中主要涉及的管理实体是 block_device、hd_struct 和 gendisk。与操作相关的是 request 和 bio 结构，另外图中还涉及与 buffer/page cache 的关系。buffer/page cache 是将文件系统等非内存空间的数据存放在内存页面中，所以在系统结构上通过内存管理的 page 以及地址转换管理 address_space（可以根据不同的内容重载为不同的管理操作）实现这部分的管理，块设备的数据单元块在内存中通过 buffer_head 进行管理，一个物理页中会存放多个块，而物理页的管理实体 page 会通过链表对 buffer head 进行管理，进而管理其中存放的块。而 page 中的 private 和标识共同表示其中管理的是 buffer_head 链表。buffer_head 和 page 是通过 attach_page_buffers 来进行关联该函数会通过 set_page_private 来设置 page 的 private 属性，这样通过 page，buffer_head 以及 address_space 就实现了 buffer/page cache 的管理。涉及块设备层和上层之间的缓冲操作也可以通过该框架实现，只要实现不同的 address_space 操作即可。

物理的块设备（如硬盘）通常可以进行分区，而不同分区在用户看来是不同的设备。硬盘本身同样是设备，这样实际就有设备层次，需要在文件系统层面能够了解这种设备层次，是

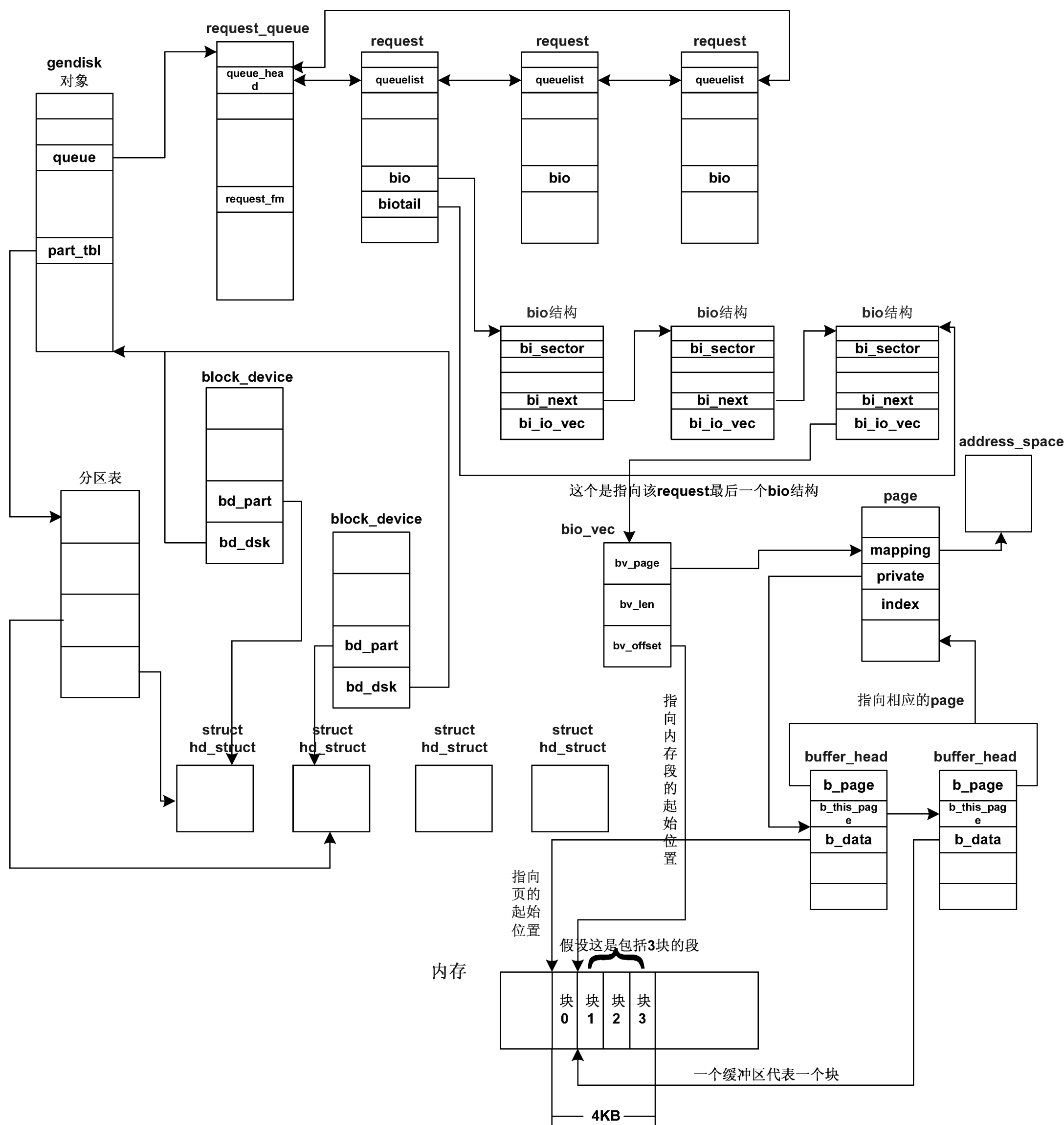


图 5-22　块设备层各实体及系统关系框图

设备管理的需求，否则用户无法了解某个分区及物理磁盘的关系。这就要求块设备对应的文件系统相关的管理实体能够表现这种层次关系。同一个物理磁盘，相关的操作没有差别，与之上的文件系统没有关系，所以驱动管理实体并不需要每个分区有一个，为了实现数据的批处理操作相应的操作数据需要以链表的形式存在。再者在块设备层与文件系统的接口管理实体中直接表示硬件的分区信息是不合适的，所以也要一个实体进行物理分区的管理，分区本身就是实际的逻辑设备，所以也会将设备模型相关的管理归入其中。以上的管理都是通过管理实体 block_device、hd_struct 和 gendisk 来实现的。它们之间各种关系的细节如图 5-23 所示。

从图 5-23 可见，文件系统中的设备层次由 block_device 来实现，驱动的管理实体通过 gendisk 来进行管理，hd_struct 进行物理的分区管理，同时为设备模型的接口。这样各种结

构的功能就明确了，接下来看看细节。

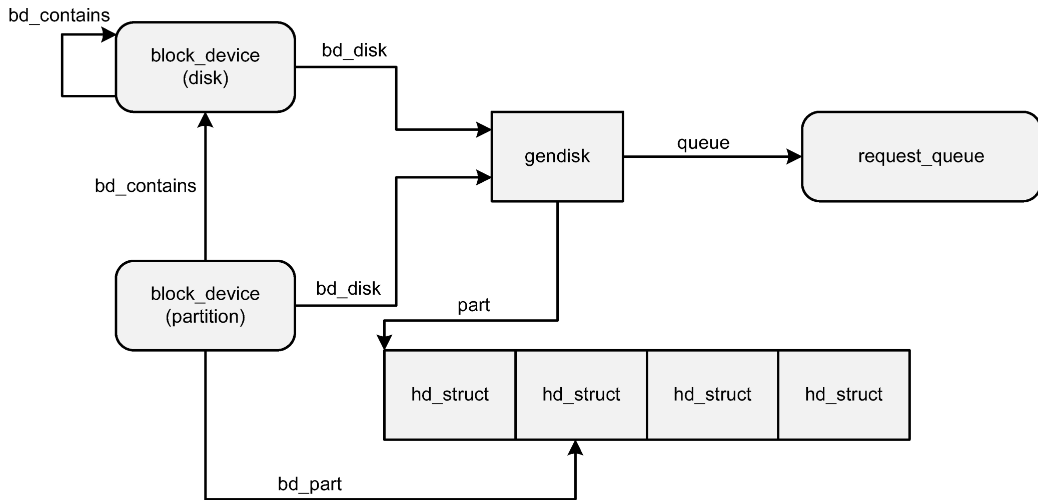


图 5-23　块设备内部管理实体关系框图

先看看 block_device 的主要信息：

```
struct block_device {
    //设备号
    dev_tb              d_dev;              /* not a kdev_t - it's a search key */
    //文件系统相关关联接口
    struct inode *      bd_inode;           /* will die */
    struct super_block * bd_super;
    //打开次数
    int                 bd_openers;
    struct mutex        bd_mutex;           /* open/close mutex */
    struct list_head    bd_inodes;
    void *              bd_claiming;
    //块设备会和 VFS 以及具体文件系统中的管理实体关联
    //这里 bd_holder 表明某个时间与哪个实体关联
    void *              bd_holder;
    int                 bd_holders;
#ifdef CONFIG_SYSFS
    struct list_head    bd_holder_list;
#endif
    //使用此属性表示设备的层次,子分区指向容器的设备 block_device
    struct block_device *bd_contains;
    //块大小
    unsigned            bd_block_size;
    //物理分区信息
    struct hd_struct *  bd_part;
```

```
	/* number of times partitions within this device have been opened. */
	unsigned			bd_part_count;
	int				bd_invalidated;
	//驱动管理实体
	struct gendisk *		bd_disk;
	//系统整体的 block_device 管理
	struct list_head		bd_list;
...
};
```

从其中的属性可见与 VFS 以及具体的文件系统都会有关联，这点不难理解。设备本身可以作为文件提供给用户，另外块设备还是具体的文件系统的承载，这两点都要体现，就都需要关联彼此了。

接下来看看 hd_ struct 的主要信息：

```
struct hd_struct {
	//物理分区信息
	sector_t start_sect;
	sector_t nr_sects;
	sector_t alignment_offset;
	unsigned int discard_alignment;
	//设备模型关联信息
	struct device __dev;
	struct kobject *holder_dir;
	int policy, partno;
	//物理分区表原信息
	struct partition_meta_info *info;
...
};
```

其中不仅包含了物理分区的信息还与设备模型有关联，毕竟分区就是对应着物理的设备。相应的设备模型的管理主要是在设置__dev 上，在 add_partition 中的设置如下：

```
device_initialize(pdev);
pdev->class = &block_class;
pdev->type = &part_type;
pdev->parent = ddev;

err = blk_alloc_devt(p, &devt);
if(err)
	goto out_free_info;
pdev->devt = devt;
```

可见相关操作时初始化 device 实体并分配设备号，其中 device 关联特定的 part_type 类型，生命周期结束时可以释放相关的空间。整个磁盘又是如何与设备模型相关联呢？这就需要看看 gendisk：

```
struct gendisk {
    //设备号相关
    int major;                  /* major number of driver */
    int first_minor;
    int minors;                 /* maximum number of minors, =1 for disks that can't be
partitioned. */

    char disk_name[DISK_NAME_LEN];/* name of major driver */
    char *(*devnode)(struct gendisk *gd, mode_t *mode);

    //实际的分区设备表,其中包含 hd_struct
    struct disk_part_tbl __rcu *part_tbl;
    //用于表示 disk 的物理信息以及与设备模型关联
    struct hd_struct part0;

    //物理 disk 的控制操作接口
    const struct block_device_operations *fops;
    //运行时设备需要操作数据的队列及操作接口,主要是数据流的操作
    struct request_queue *queue;
    void *private_data;

    int flags;
    struct device *driverfs_dev;    // FIXME: remove
    struct kobject *slave_dir;
...
};
```

可见其中嵌入了 struct hd_struct part0；这样既有了整体 disk 的物理信息又可以与设备模型相关联。在 alloc_disk 中会调用 alloc_disk_node，其中包含如下代码：

```
disk_to_dev(disk)->class = &block_class;
disk_to_dev(disk)->type = &disk_type;
device_initialize(disk_to_dev(disk));
```

这里设置的设备模型相关类型为 disk_type，可以对 disk 进行生命周期管理，在必要的时候可以释放其中分区的信息。

3. 驱动管理及块设备驱动流程

块设备层的驱动管理工作同样是由 kobj_map 来实现的，kobj_map 中的内容之前在字符设备中已经进行了介绍，下面看看块设备层是如何使用它完成驱动管理的。

驱动管理要涉及设备号的范围及管理实体，块设备层将其封装为 blk_register_region，细节如下：

```
void blk_register _region( dev_t devt, unsigned long range, struct module  * module,
                 struct kobject  * ( * probe) ( dev_t, int  * , void  * ),
                 int( * lock) ( dev_t, void  * ), void  * data)
{
    kobj_map( bdev_map, devt, range, module, probe, lock, data);
}
```

下面来看看具体的管理实体是什么？相关在 add_disk 中的实现如下：

```
void add_disk( struct gendisk  * disk)
{
    struct backing_dev_info  * bdi;
    dev_t devt;
    int retval;
…
    //分配 disk 的设备号
    retval = blk_alloc_devt( &disk -> part0, &devt);
    if( retval)
        { WARN_ON(1);
        return;
    }
    disk_to_dev( disk) -> devt = devt;

    / *  -> major and  -> first_minor aren' t supposed to be
      * dereferenced from here on, but set them just in case.
      * /
    //记录设备号
    disk -> major = MAJOR( devt);
    disk -> first_minor = MINOR( devt);

    / *  Register BDI before referencing it from bdev  * /
    bdi = &disk -> queue -> backing_dev_info;
    bdi_register_dev( bdi, disk_devt( disk));

    //这里注册驱动管理实体
    blk_register_region( disk_devt( disk), disk -> minors, NULL,
                                exact_match, exact_lock, disk);
    //向 sys 文件注册相关信息
    register_disk( disk);
    //向 sys 文件注册数据流的操作信息,提供操作接口
```

```
        blk_register_queue(disk);

        retval = sysfs_create _link(&disk_to_dev(disk) ->kobj, &bdi ->dev ->kobj,
                                   "bdi");
        WARN_ON(retval);
}
```

从代码分析可见，驱动管理实体是 gendisk，在注册驱动管理实体的同时还向 sys 文件系统注册一些属性，以便运行时可以进行相关设置。

块设备驱动管理的子设备号由具体的块设备进行设置，因为这和其能力相关，块设备层提供了 alloc_disk 接口进行相关操作。

驱动管理实体 gendisk 的细节如图 5-24 所示。

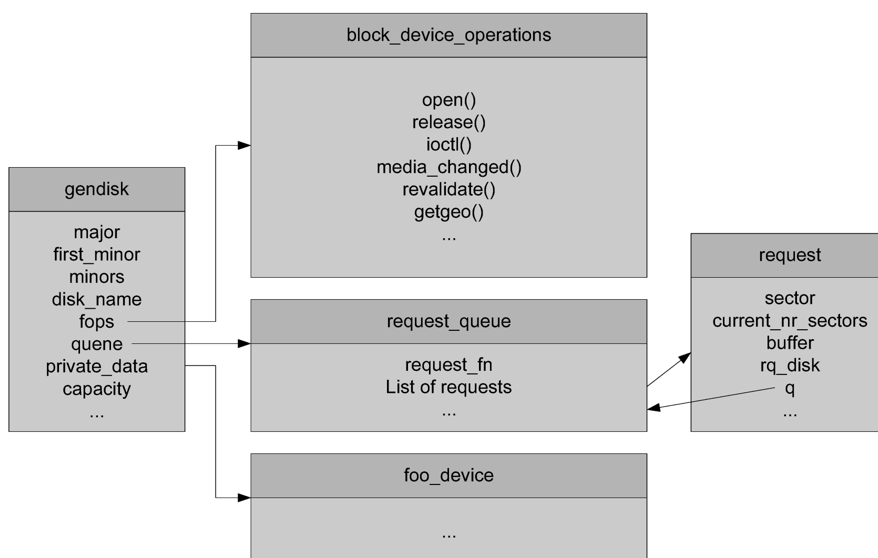


图 5-24　块设备驱动管理实体

从图 5-24 可见，其中主要包括设备的控制操作接口 block_device_operations，以及与设备数据流相关的数据队列及操作。block_device_operations 中主要是对设备打开、关闭，以及换盘等控制操作的接口。这样设计的原因是：块设备是文件系统的物理基础，而物理文件系统不会调用 VFS 中的文件接口（这样做使得层次混乱），所以相关的控制操作通常会在块设备层中进行封装，比如 blkdev_get 就会进行设备 open 的操作，这样可以为上层各种需求提供统一的接口。

块设备数据流请求的具体关系以一个例子进行说明，如图 5-25 所示。

图 5-25 中的一次操作的请求是读取连续 16KB 的数据，由于 Linux 内核中将 sect 定义为 512 B，所以是 32 个 sect。相应的请求拆成两个 bio 进行操作，每个 bio 为 8KB 的操作，对应到内存的两个 page，相应分配 bio_vec 的数组为 2 会关联到两个物理页面的管理实体。实际的操作就是在页面与 bio 之间进行，通常这些块设备 DMA 的操作通过散列式 DMA 的方式实现，以提高整体的性能。

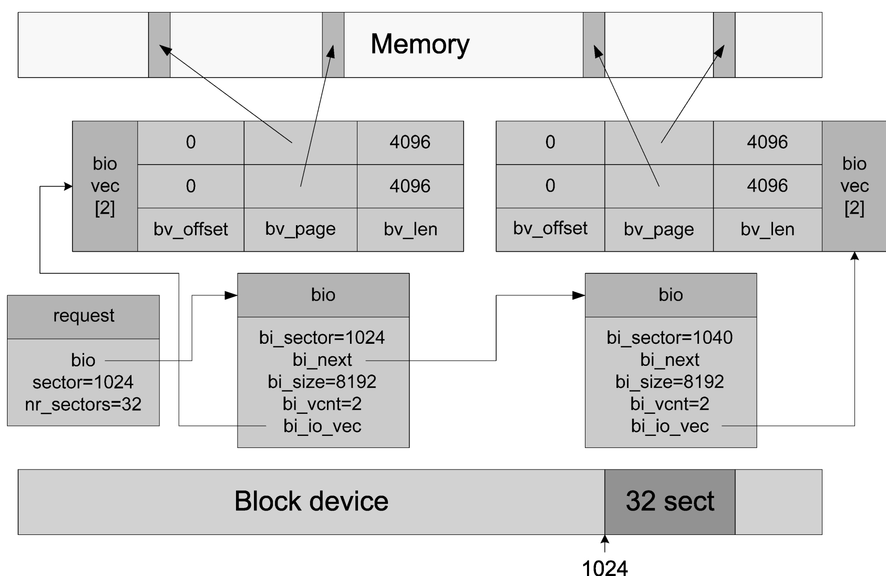


图 5-25　块设备数据请求例子

针对块设备的数据操作流程，块设备层统一提供了操作单元 bio，相应的起始操作接口是 submit_bio，从 submit_bio 到请求队列的流程如图 5-26 所示。

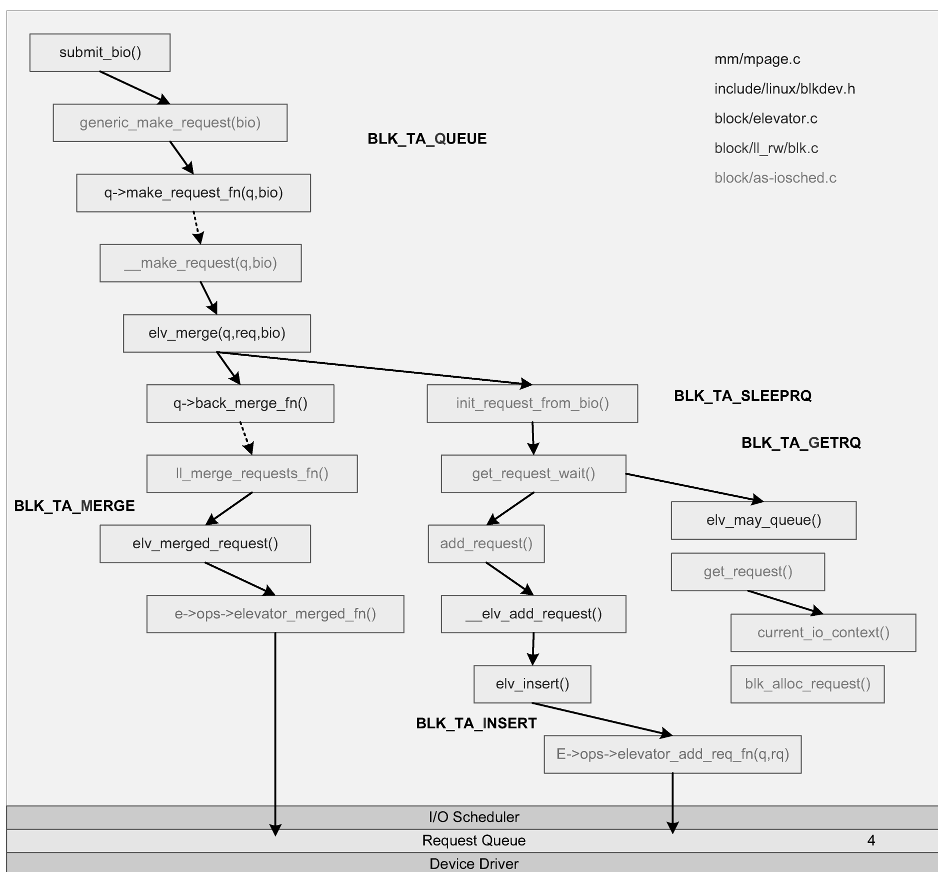


图 5-26　submit_bio 到请求队列流程

从图 5-26 可见，在真正处理请求之前是可以对请求进行合并等操作的，这样保证最终执行的请求尽可能的集中，以减少如机械硬盘等有机械件的块设备的机械操作。

通常情况下具体块设备相应的请求处理流程如图 5-27 所示。

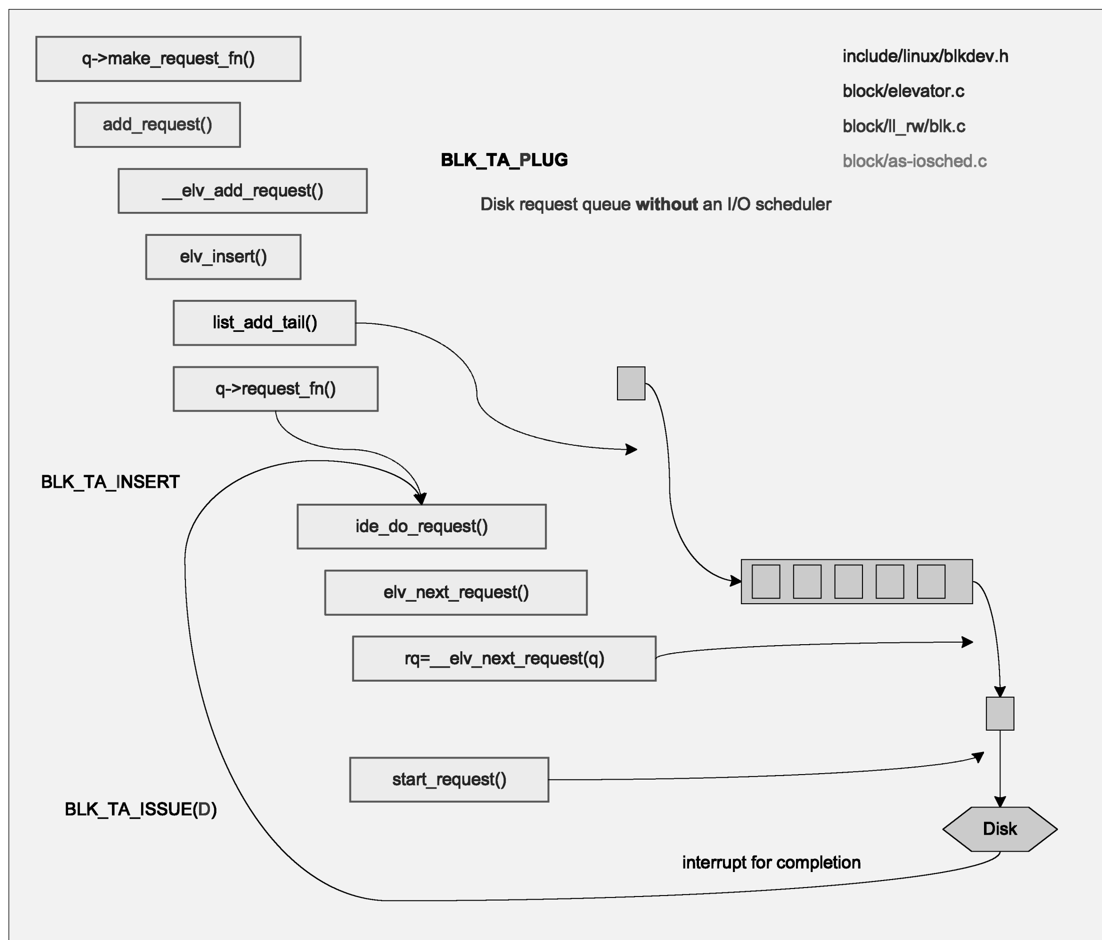


图 5-27　一般块设备请求处理流程

从图 5-27 可见，上层下发的请求是完全按照顺序进行处理的，其中请求队列中的 request_fn 通常是唤醒具体块设备的驱动进程进行实际的请求处理，这样将上层与驱动分开在不同的执行实体执行，可以通过对数据进行批处理操作，从而提高整体的性能。

从系统的角度，IO 性能除了吞吐量外，还要考虑不同应用 IO 请求的响应延时。如果只是按照顺序进行块设备的请求处理并不能很好地满足各种响应延时的需求，所以需要在请求执行过程中加入调度的机制，就是 IO scheduler，来提升整体的性能。由于这些都是对实际操作请求的调整，所以都被包含在 request_queue 中，不同的 IO scheduler 方式由 struct elevator_type 来定义，主要的操作接口包含在 struct elevator_ops 中。request_queue 如果使用 IO Scheduler 则逻辑上有两套 queue，一个是 IO scheduler 使用的 queue；另一个是物理驱动操作的 queue。具体的流程如图 5-28 所示。

从图 5-28 可见，真正对请求的操作顺序是会依据之前的 IO 操作重新调整顺序的，作为

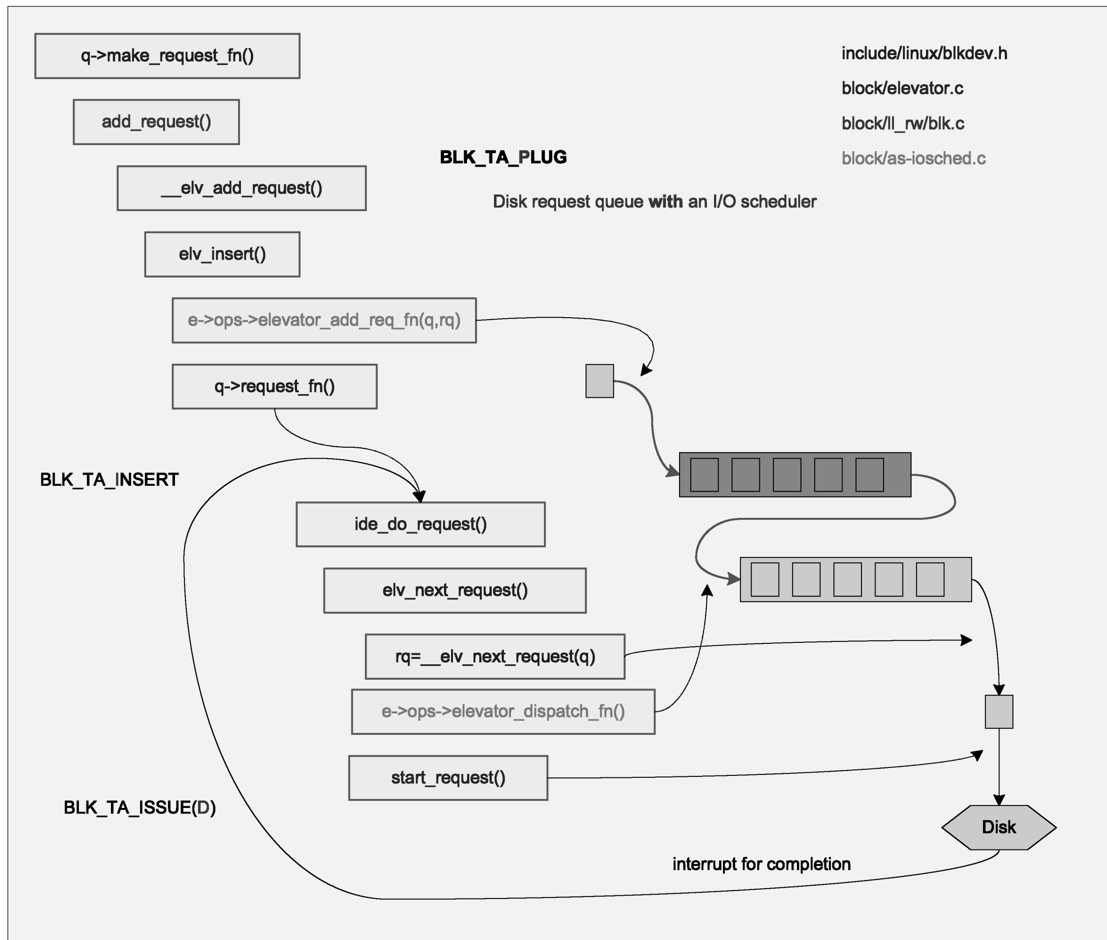


图 5-28　带 io scheduler 的块设备请求处理流程

内核块设备层可以实现一个反馈系统，从而尽力满足所有用户的性能需求，提高整体的 IO 性能。当然不会有对所有情况都满足的调度方法，所以 Linux 内核提供了很多种 IO scheduler 方法，如 CFQ、Deadline 等以适应不同的需求。系统可以通过 elevator_switch 来进行不同调度算法之间的切换。由于 IO scheduler 是与算法紧密相关的这里就不进行详细的分析。

4. 块设备驱动初始化

针对块设备驱动，块设备层提供了很多接口包括如何操作请求队列（start，stop，延时操作等），还提供了默认生成 request 的接口__make_request，但是只有一个接口 request_fn 需要由驱动提供，前面提到了该接口是具体唤醒驱动请求处理的函数，也是和具体驱动关系最紧密的部分，相关的初始化就是要将驱动特定的 request_fn 接口加入 request_queue 中，Linux 内核通过 blk_init_queue 生成包含驱动特定接口的 request_queue 来实现该功能。具体驱动的初始化流程如图 5-29 所示。

从图 5-29 可见，对于具体的块设备驱动中最重要的两部分就是 request_fn 和中断。request_fn 负责下发请求并唤醒处理逻辑；中断则负责处理之后的流程唤醒。两者完整的结合实现整体数据操作。

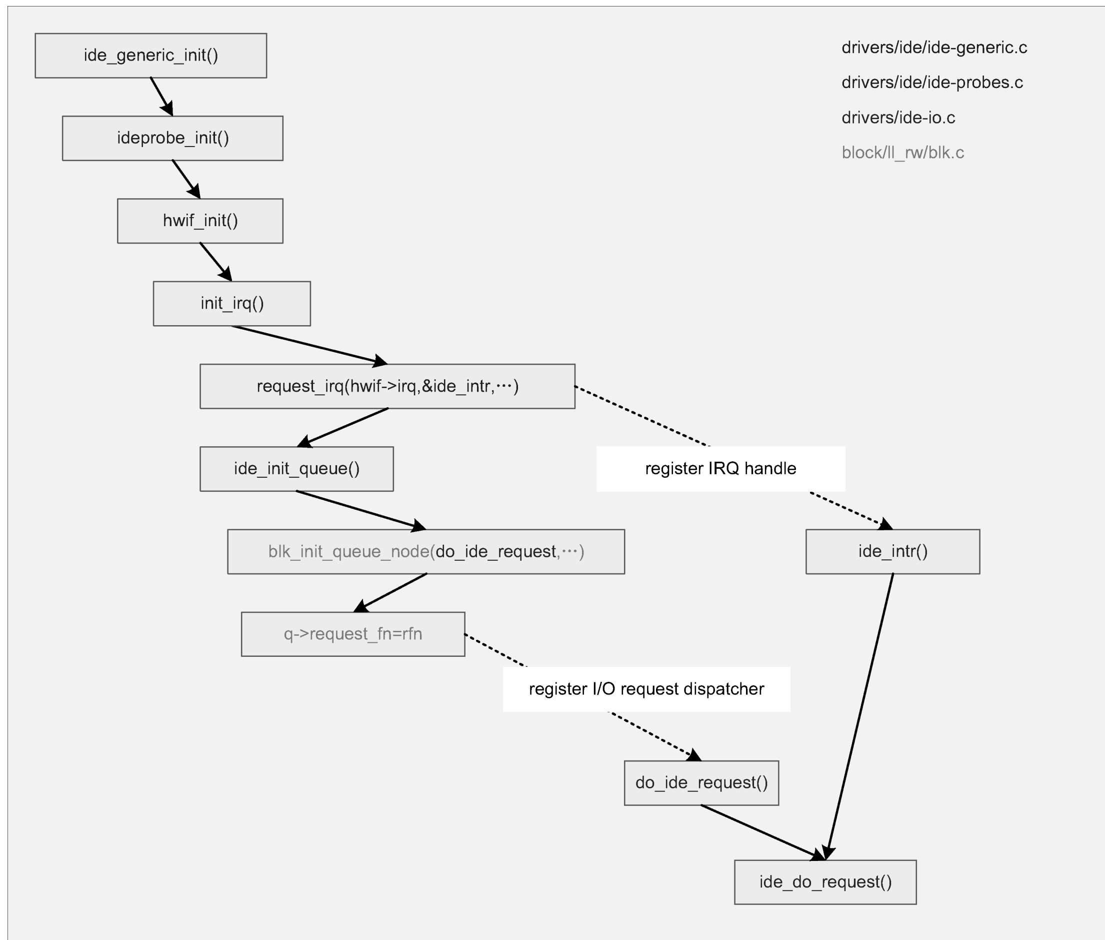


图 5-29　具体驱动的初始化流程

在块设备框架层，具体的块设备驱动初始化一般按照以下流程实现：首先需要通过 alloc_disk 来分配驱动管理实体 gendisk；然后将 blk_init_queue 生成的设备自身 request_queue 加入到 gendisk 中；最后通过 add_disk 来将驱动管理实体加入系统进行管理。这样系统就可以通过驱动进行操作了。

5.4.3　块设备子类型

块设备的类型同样是通过设备号进行管理的，块设备注册设备号的接口是 register_blkdev，接口定义如下：

```
int register_blkdev(unsigned int major, const char * name)
```

可见块设备并没有通过子设备号进行分类，毕竟块设备类型的数量与字符设备还是有差距的。所以了解系统块设备的分类只需要看/proc/devices 即可。细节如下：

```
Block devices:
  1 ramdisk
259 blkext
  7     loop
  8     sd
  9     md
 11     sr
 65     sd
 66     sd
 67     sd
 68     sd
 69     sd
 70     sd
 71     sd
128     sd
129     sd
130     sd
131     sd
132     sd
133     sd
134     sd
135     sd
251     zram
252     device - mapper
253     virtblk
254     mdp
```

从中可见有很多 sd，sd 是硬盘的总称，有各种各样的硬盘如 IDE、SCSI 等。具体分类信息可以通过 Documentation/devices. txt 文件获得。后面会对部分设备进行详细分析。

5. 5　电源管理

5. 5. 1　电源管理特点和需求

Linux 内核在设计之初并没有考虑电源管理，但是随着节能需求不断提高，特别是终端类产品迅速发展，对于电源管理的需求越来越强烈，Linux 内核也提供了电源管理功能。

电源管理相关的硬件知识之前已经有了介绍，对软件来说基本上就是利用硬件的能力实现系统“能睡就睡”，设备能降低频率就降低频率，另外还要满足用户主动的电源管理需求。进行这些操作的前提是要能保证系统和设备正常运行。

5.5.2 电源管理核心框架介绍

Linux 内核的电源管理部分在 X86 架构上有 ACPI（高级配置与电源管理接口），但是广大的嵌入式设备并不支持 ACPI，而对嵌入式设备来说电源管理的需求更强烈，所以急需相关的功能实现。考虑到电源管理的需求涉及处理器和各种设备，一方面是处理器尽可能减少功耗，另一方面是设备尽可能减少功耗。减少功耗的功能框架各种各样，一种功能框架是不能满足电源管理的需求的。所以 Linux 内核将电源管理分成不同的功能模块来实现，这样通过各个功能模块整体工作来满足电源管理的需求。Linux 内核电源管理功能如图 5-30 所示。

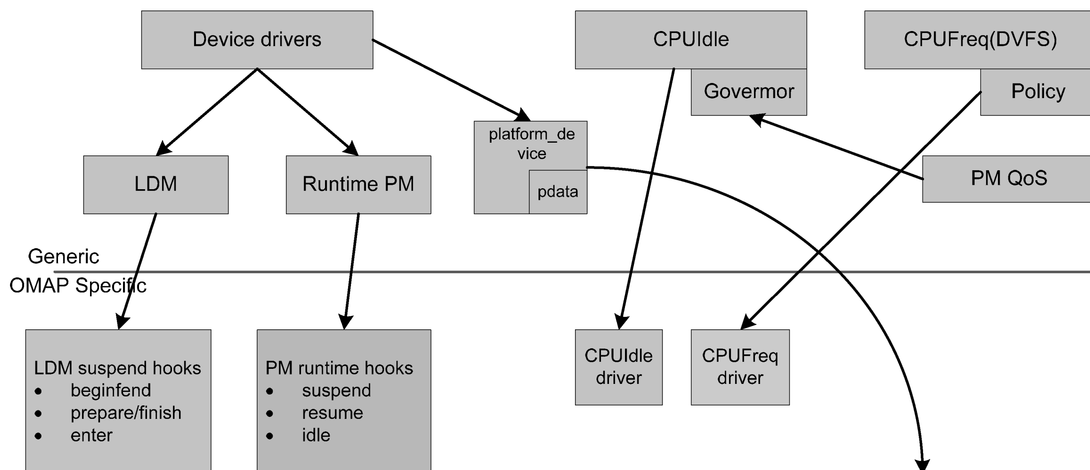


图 5-30　Linux 电源管理各个功能

从图 5-30 可见 Linux 内核的电源管理功能有与处理器相关的 CPUIdle 和 CPUFreq，也有与设备相关的 runtime pm，另外还有与整个系统待机时 SLM（standby leakage management）相关的低功耗电源管理功能。但并不是只有这些功能，在时间管理中的 dynamic tick 也属于降低功耗的功能。下面分别对这些功能框架进行介绍。

1. CPUIdle

在初始化的时候有过介绍，系统在没有任务需要调度的时候会执行 cpu_idle，使处理器处于休息状态，其中系统提供了 pm_idle 接口实现各种 idle 的省电功能。CPUIdle 会将其改为自己的接口 cpuidle_idle_call，这样在内核进入 idle 时就会调用相应的接口，从而进入合适的状态。接下来分析一下相关的功能：

```
static void cpuidle_idle_call(void)
{
    //首先获得处理器特定的设备包含属性和接口信息
    struct cpuidle_device *dev = __get_cpu_var(cpuidle_devices);
    struct cpuidle_state *target_state;
    int next_state;
```

```
…
    //如果需要进行设备进入 idle 的准备操作
    if(dev->prepare)
        dev->prepare(dev);

    /* ask the governor for the next state */
    //通过设置的主管逻辑以及设备的 QoS 选择合适的状态
    next_state = cpuidle_curr_governor->select(dev);
    //如果发现需要调度,则不应该睡眠了,由于进入时关中断,所以这里开中断
    if(need_resched()) {
        local_irq_enable();
        return;
    }

    target_state = &dev->states[next_state];

    /* enter the state and update stats */
    //进入睡眠操作并进行统计
    dev->last_state = target_state;
    dev->last_residency = target_state->enter(dev, target_state);
    if(dev->last_state)
        target_state = dev->last_state;

    target_state->time += (unsigned long long)dev->last_residency;
    target_state->usage++;

    /* give the governor an opportunity to reflect on the outcome */
    //通知主管相关操作,使得其算法得到相关信息
    if(cpuidle_curr_governor->reflect)
        cpuidle_curr_governor->reflect(dev);
    trace_power_end(smp_processor_id());
}
```

分析代码可见，具体的驱动要提供多种状态，这些状态是和 QoS 相关的。所谓 QoS 就是指对于 idle 来说不是简单地睡就可以了，要考虑醒来工作是否可以正常，例如进入不同的状态系统睡眠和唤醒延时是不同的，而 idle 只是短时间进入低功耗，相关的任务更重要，不能为了进入功耗更低的状态而影响正常的工作，这就需要 QoS 机制。在标记状态的同时要标记唤醒的时间，由 governor 来根据当前的系统状态选择合适的低功耗状态进入，既要休息好又不能影响工作。CPUIdle 睡眠示意如图 5-31 所示。

从图 5-31 可见，系统要考虑进入哪种状态与能够睡眠的时间、sleep 和 wakeup 延时都是相关的，能睡眠的时间越长越应该进入更高级别的低功耗状态。menu governor 是通过每个处理器的下一次 tick 的时间来决定可进入低功耗的时间，从而进行选择。

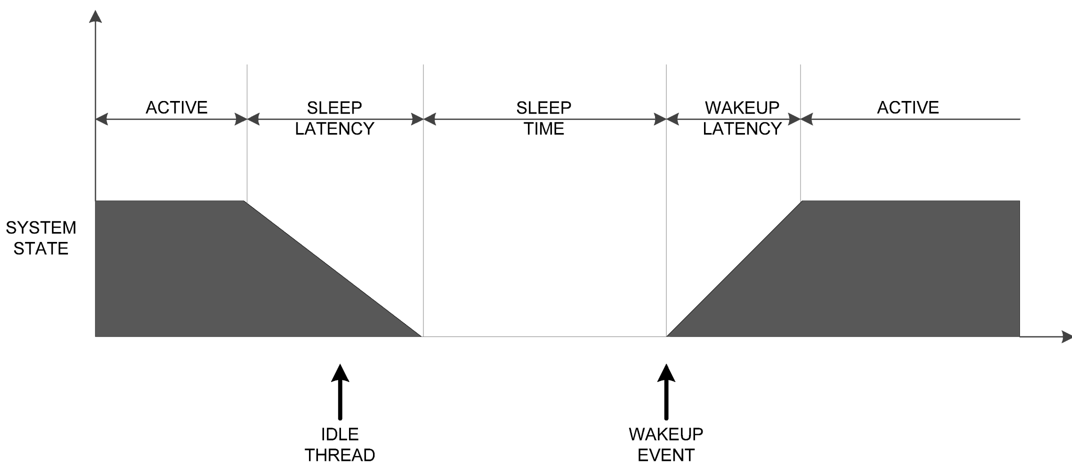


图 5-31　CPUidle 电源管理示意图

具体的设备驱动会在 SoC 电源管理部分进行讲解。

2. CPUFreq

CPUFreq 就是实现处理器的频率变化，其中会涉及 DVFS 来降低功耗，相关的硬件原理之前已经介绍。这里对 Linux 内核的软件实现进行分析。

在介绍无关性实现的时候，已经介绍了 CPUFreq 的框架（见图 3－21），在 CPUFreq 中不管根据什么原则，是监视系统状态也好还是用户设置也好，在选择了相应的频率等级后最终都是通过__cpufreq_driver_target 来进行频率设置的，其中会调用具体的设备 target 接口进入相应的频率等级。具体的驱动同样会在 SoC 电源管理部分进行讲解。

这里看看 governor 的实现，主要介绍 ondemand governor。流程图如图 5-32 所示。

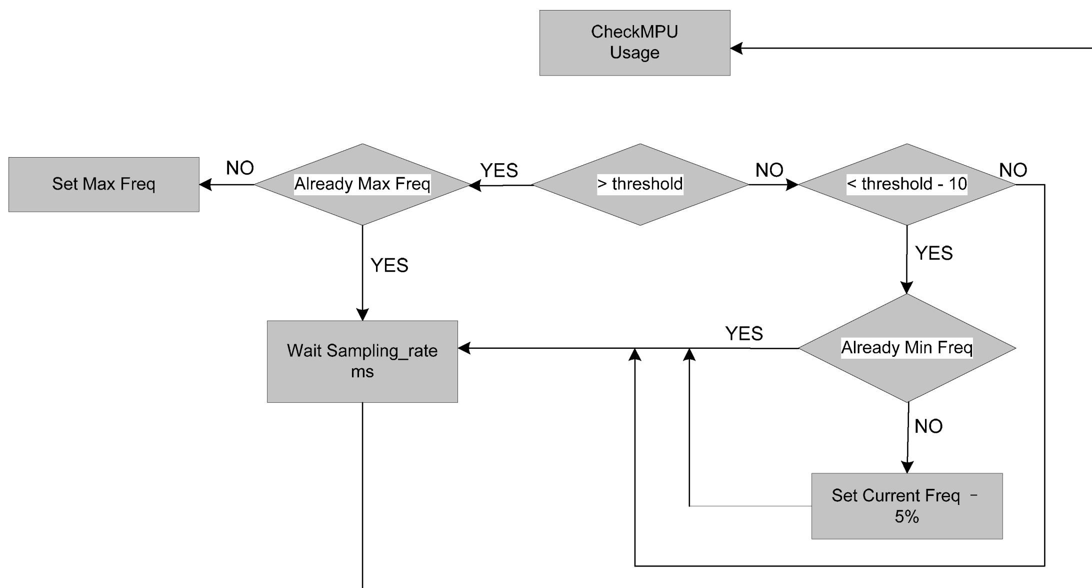


图 5-32　ondemand 流程图

从图 5-32 可见，ondemand governor 主要是监控 CPU 的使用率是否超过一定的值，如果是就提高主频，如果使用率很低就降低主频，从而实现根据使用率的动态调节以达到按需使用的功能。相关的参数配置在/sys/devices/system/cpu/cpu0/cpufreq/目录下，可以根据需要进行设置。

3. SLM

这两部分功能都和设备相关。首先来看看设备模型相关的初始化部分，如图 5-33 所示。

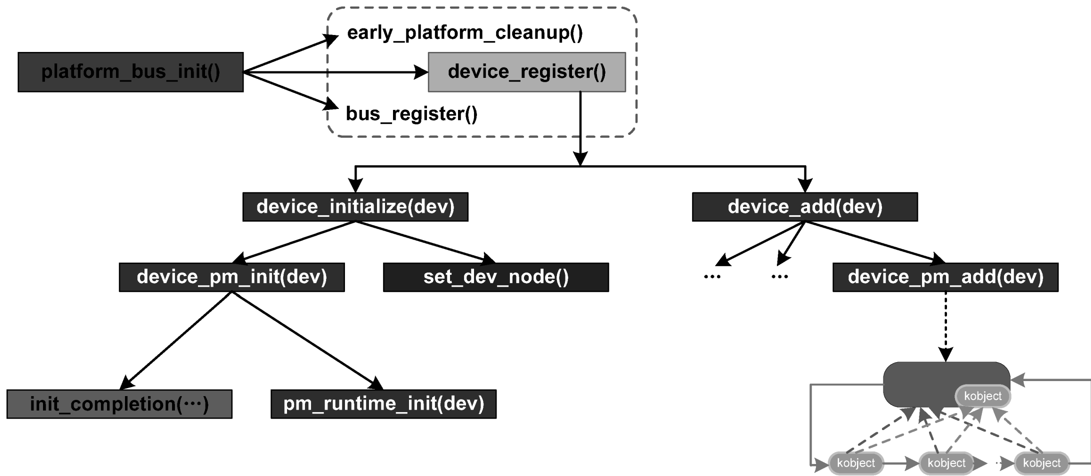


图 5-33　设备模型电源管理相关初始化

从图 5-33 可见，其中涉及了设备 Runtime pm 的相关初始化，但是 SLM 的操作并不明显。其实设备的 SLM 相关操作已经隐含在 device 添加过程 device_pm_add 中，思考一下，SLM 操作时要进入某种待机状态，其中会涉及所有的设备，自然将其放入系统的设备增加过程中。这样系统的 SLM 操作就可以通过设备模型统一管理了。下面来看看细节：

```
void device_pm_add(struct device *dev)
{
    pr_debug("PM: Adding info for %s:%s\n",
        dev->bus ? dev->bus->name : "No Bus",
        kobject_name(&dev->kobj));
    mutex_lock(&dpm_list_mtx);
    if(dev->parent) {
        if(dev->parent->power.status >= DPM_SUSPENDING)
            dev_warn(dev, "parent %s should not be sleeping\n",
                dev_name(dev->parent));
    } else if(transition_started) {
        /*
         * We refuse to register parentless devices while a PM
         * transition is in progress in order to avoid leaving them
```

```
                * unhandled down the road
                */
            dev_WARN(dev, "Parentless device registered during a PM transaction\n");
        }

        //这里主要是将设备加入 dpm_list 设备功耗管理列表进行管理
        list_add_tail(&dev->power.entry, &dpm_list);
        mutex_unlock(&dpm_list_mtx);
    }
```

既然所有的设备都会通过 device_add 添加到设备模型中进行管理，那么相应的也会加入到设备功耗管理列表中进行管理。

那么系统如何进入待机状态呢？这就要看 sys 文件系统了，如图 5-34 所示。

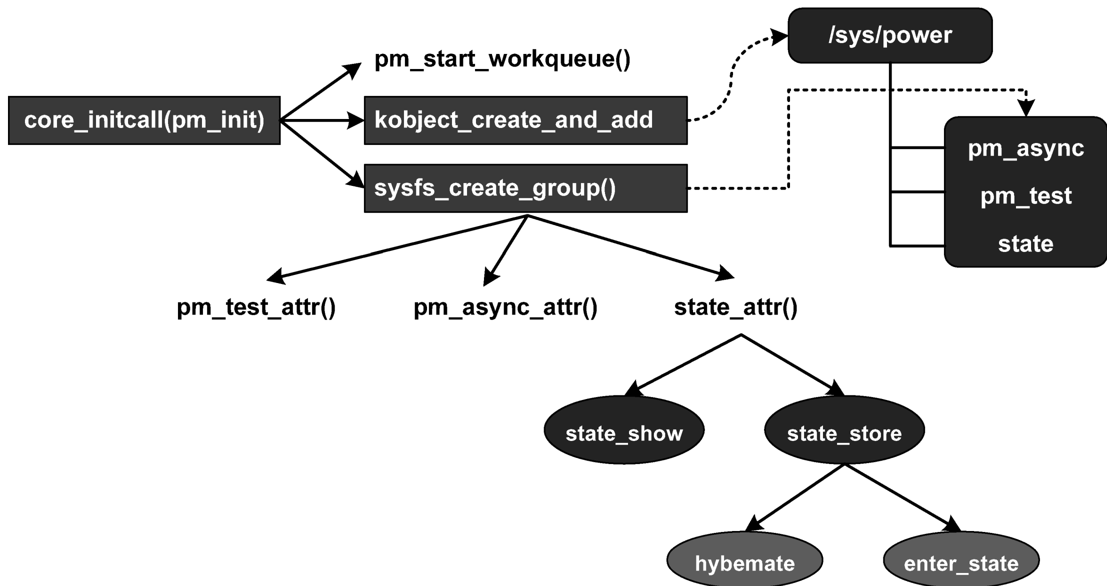


图 5-34　待机控制框图

从图 5-34 可见，/sys/power 目录下的 state 文件用于设置具体的待机状态。通过 enter_state 即可进入相应的状态，最终要进入待机 suspend 状态才会调用 dpm_suspend_noirq。下面来看看实现细节：

```
int dpm_suspend_noirq(pm_message_t state)
{
    struct list_head list;
    //记录操作时间
    ktime_t starttime = ktime_get();
    int error = 0;
```

```
        INIT_LIST_HEAD(&list);
        suspend_device_irqs();
        mutex_lock(&dpm_list_mtx);
        //遍历设备功耗管理列表的所有设备
        while(!list_empty(&dpm_list)) {
        //从尾部开始遍历,这样保证按层次执行
        struct device *dev = to_device(dpm_list.prev);
            //保证设备不被释放
            get_device(dev);
            mutex_unlock(&dpm_list_mtx);

            //使设备进入相应的状态
            error = device_suspend_noirq(dev, state);

            mutex_lock(&dpm_list_mtx);
            if(error) {
                pm_dev_err(dev, state, " late", error);
                put_device(dev);
                break;
            }
            //记录设备状态
            dev->power.status = DPM_OFF_IRQ;
            //将正确操作的设备移入临时的列表
            if(!list_empty(&dev->power.entry))
                list_move(&dev->power.entry, &list);
            //操作完计数减少
            put_device(dev);
        }
        //将正常进入 suspend 的设备合并进设备功耗管理列表
        list_splice_tail(&list, &dpm_list);
        mutex_unlock(&dpm_list_mtx);
        //如果有错误,说明进不了待机状态,则需要恢复系统以便正常执行
        if(error)
            dpm_resume_noirq(resume_event(state));
        else
            dpm_show_time(starttime, state, "late");
        return error;
    }
```

从代码中可见，是想让所有的设备都进入待机状态，下面来看看通过 device_suspend_noirq 实现，设备模型对具体的设备进入相应待机状态的操作：

```
static int device_suspend_noirq(struct device *dev, pm_message_t state)
{
    int error = 0;

    //检查是否是抽象功能类型设备,如果是调用相关接口
    if(dev -> class && dev -> class -> pm) {
        pm_dev_dbg(dev, state, "LATE class ");
        error = pm_noirq_op(dev, dev -> class -> pm, state);
        if(error)
            goto End;
    }

    //检查具体的设备类型(会依据 class 和 bus 有不同情况),并进行相关操作
    if(dev -> type && dev -> type -> pm) {
        pm_dev_dbg(dev, state, "LATE type ");
        error = pm_noirq_op(dev, dev -> type -> pm, state);
        if(error)
            goto End;
    }

    //检查是否是挂在总线 bus 的设备,如果是进行相关操作
    if(dev -> bus && dev -> bus -> pm) {
        pm_dev_dbg(dev, state, "LATE ");
        error = pm_noirq_op(dev, dev -> bus -> pm, state);
    }

End:
    return error;
}
```

从具体实现中可见，其操作逻辑是按照之前设备模型介绍，依据设备具体的情况进行的。对具体设备的电源管理实现，将在设备驱动中进行详细分析。SoC 内部的接口设备通常都是 platform_device，所以会执行 bus 的 pm 操作，具体如下：

```
static const struct dev_pm_ops platform_dev_pm_ops = {
    .prepare = platform_pm_prepare,
    .complete = platform_pm_complete,
    .suspend = platform_pm_suspend,
    .resume = platform_pm_resume,
    .freeze = platform_pm_freeze,
    .thaw = platform_pm_thaw,
    .poweroff = platform_pm_poweroff,
```

```
        .restore = platform_pm_restore,
        .suspend_noirq = platform_pm_suspend_noirq,
        .resume_noirq = platform_pm_resume_noirq,
        .freeze_noirq = platform_pm_freeze_noirq,
        .thaw_noirq = platform_pm_thaw_noirq,
        .poweroff_noirq = platform_pm_poweroff_noirq,
        .restore_noirq = platform_pm_restore_noirq,
        .runtime_suspend = platform_pm_runtime_suspend,
        .runtime_resume = platform_pm_runtime_resume,
        .runtime_idle = platform_pm_runtime_idle,
};
```

而在 suspend 操作中会执行 platform_pm_suspend_noirq，细节如下：

```
int __weak platform_pm_suspend_noirq(struct device *dev)
{
    struct device_driver *drv = dev->driver;
    int ret = 0;

    if(!drv)
        return 0;

    if(drv->pm) {
        if(drv->pm->suspend_noirq)
            ret = drv->pm->suspend_noirq(dev);
    }

    return ret;
}
```

从中可见，具体的是调用相应的 platform_driver 接口函数，这样转换好处是使得 platform_device 只维护资源，而 driver 则是可以复用的操作接口。

再来看看使整个系统进入待机的接口：

```
static int suspend_enter(suspend_state_t state)
{
    int error;

    //系统级的 suspend 接口通常由具体的芯片设置相关接口
    if(suspend_ops->prepare) {
        error = suspend_ops->prepare();
        if(error)
```

```
            goto Platform_finish;
    }

    //对所有的普通设备进行 suspend 操作
    error = dpm_suspend_noirq(PMSG_SUSPEND);
    if(error) {
        printk(KERN_ERR "PM: Some devices failed to power down\n");
        goto Platform_finish;
    }

    //继续系统级的操作
    if(suspend_ops -> prepare_late) {
        error = suspend_ops -> prepare_late();
        if(error)
            goto Platform_wake;
    }

    if(suspend_test(TEST_PLATFORM))
        goto Platform_wake;

    //将非启动的处理器 suspend
    error = disable_nonboot_cpus();
    if(error || suspend_test(TEST_CPUS))
        goto Enable_cpus;

    //体系结构上可以关中断
    arch_suspend_disable_irqs();
    BUG_ON(! irqs_disabled());

    //sysdev 进行 suspend 操作
    error = sysdev_suspend(PMSG_SUSPEND);
    if(! error) {
        if(! suspend_test(TEST_CORE) && pm_check_wakeup_events()) {
            //这里整个系统进入相应的低功耗状态,整个系统待机只
            //响应唤醒事件
            error = suspend_ops -> enter(state);
            events_check_enabled = false;
        }
        //这里系统已经唤醒,先要恢复 sysdev
        sysdev_resume();
    }
```

```
        //恢复流程
        arch_suspend_enable_irqs();
        BUG_ON(irqs_disabled());

    Enable_cpus:
        enable_nonboot_cpus();

    Platform_wake:
        if(suspend_ops->wake)
            suspend_ops->wake();

        dpm_resume_noirq(PMSG_RESUME);

    Platform_finish:
        if(suspend_ops->finish)
            suspend_ops->finish();

        return error;
    }
```

以上分析主要是进入待机状态最后阶段的操作流程，而要使得设备和系统进入待机状态，只提供一个操作接口其实现的复杂度就太高了，另外由于设备层次以及设备操作的复杂性，在进入待机状态之前需要进行必要的保护，所以电源管理操作就应该提供不同阶段的操作接口，正如在 dev_pm_ops 中所见的各种函数接口。具体的 suspend 操作则分为 suspend prepare、suspend、suspend noirq 等几个不同的阶段，并在 dev_pm_ops 定义了相应的接口。为了遍历所有的设备并执行相应的操作，dpm（derice power management）也提供了相应的接口，分别是 dpm_prepare、dpm_suspend 和 dpm_suspend_noirq。这样一来所有的设备就可以通过分阶段的操作完成必要的保护操作后进入待机状态，恢复过程操作则是待机操作的逆过程。

从分析可知，整个系统的待机操作是从外围设备开始，最后到和系统电源管理最紧密的系统设备，所以从电源管理的角度设备也是分层的，外围设备就是设备模型中的 device，而系统设备是 sysdev（如 CPU、memory、核心时钟源、时间管理部分等都是通过 sysdev_register 注册的，这与最小系统所见设备抽象后的实体基本一致）。另外内核也为芯片提供了基本的 suspend_ops 待机操作接口来进行芯片级别的特殊操作。

4. runtime pm 和 device wake

下面看看设备的 runtime pm 是如何实现的。每个设备的 runtime pm 初始化工作由 pm_runtime_init 来完成，具体分析如下：

```
    void pm_runtime_init(struct device *dev)
    {
        //初始化设备相关的 runtime pm 信息
```

```
        dev->power.runtime_status = RPM_SUSPENDED;
        dev->power.idle_notification = false;

        dev->power.disable_depth = 1;
        atomic_set(&dev->power.usage_count, 0);

        dev->power.runtime_error = 0;

        atomic_set(&dev->power.child_count, 0);
        pm_suspend_ignore_children(dev, false);
        dev->power.runtime_auto = true;

        dev->power.request_pending = false;
        dev->power.request = RPM_REQ_NONE;
        dev->power.deferred_resume = false;
        dev->power.accounting_timestamp = jiffies;
        //记录设备 runtime pm 的执行实体
        INIT_WORK(&dev->power.work, pm_runtime_work);

        dev->power.timer_expires = 0;
        setup_timer(&dev->power.suspend_timer, pm_suspend_timer_fn,
                (unsigned long)dev);

        //初始化等待队列
        init_waitqueue_head(&dev->power.wait_queue);
    }
```

从代码分析中可见，主要是进行相关属性的初始化，另外初始化了用于异步执行 runtime pm 操作的 work（pm_init 中会初始化 workqueue pm_wq 来异步执行设备的 runtime pm 操作）以及相应的进行多个上下文同步的等待队列。考虑设备驱动是可以在多个上下文以及执行实体中运行，而驱动操作的设备物理上是唯一的实体，这就要求对设备进行的动态电源操作时需要进行同步。

设备的 runtime pm 操作的状态同步都是通过 dev_pm_info 来完成的，其被包含在 device 中，这样在任何上下文对设备进行 runtime pm 操作的时候都可以同步。dev_pm_info 的细节如下：

```
    struct dev_pm_info {
        //所有电源管理操作的锁
        spinlock_t              lock;
    …
    #ifdef CONFIG_PM_RUNTIME
        //进行延时操作或者自动操作需要的定时器
```

```
struct timer_list          suspend_timer;
//定时器到期的时间
unsigned long              timer_expires;
//异步操作需要的 work 会在 pm - _wq 工作队列上执行,
//在 pm_runtime_init 中初始化
struct work_struct         work;
//等待事件,用于当多个上下文对同一个设备进行相同操作时进行系统同步
wait_queue_head_t wait_queue;
//使用计数
atomic_t                   usage_count;
//active 的子设备的计数
atomic_t                   child_count;
//disable 的深度等于 0 表示可以执行 runtime pm 操作
unsigned int               disable_depth:3;
//忽略子设备的状态
unsigned int               ignore_children:1;
//表示在进行 idle 操作
unsigned int               idle_notification:1;
//表示有 suspend 操作请求
unsigned int               request_pending:1;
//当 suspend 正在执行时,需要延迟 resume 操作,设置该状态表示 resume 延迟
unsigned int               deferred_resume:1;
//表示运行时唤醒能力
unsigned int               run_wake:1;
unsigned int               runtime_auto:1;
//不执行 runtime pm 设备层面的操作接口
unsigned int               no_callbacks:1;
//自动休眠功能是否开启
unsigned int               use_autosuspend:1;
//是否在定时器到期时尝试进行自动休眠
unsigned int               timer_autosuspends:1;
//需要执行的 runtime pm 的请求
enum rpm_request           request;
//设备 runtime pm 的状态
enum rpm_status            runtime_status;
//设备当前的错误状态值,当有错误时 runtime pm 的操作无法执行
int                        runtime_error;
//自动休眠的延迟时间
int                        autosuspend_delay;
//以下是记录一些时间状态
unsigned long              last_busy;
unsigned long              active_jiffies;
```

```
    unsigned long        suspended_jiffies;
    unsigned long        accounting_timestamp;
#endif
};
```

可见其中不仅包含设备 runtime pm 的本身状态信息，还包含不同操作方式（如同步、异步）的状态。设备的 runtime pm 操作最终是通过 rpm_xxx 来执行的，xxx 代表不同的操作请求，主要是 idle、suspend 和 resume。下面以 rpm_suspend 为例来分析具体的流程：

```
static int rpm_suspend(struct device *dev, int rpmflags)
        __releases(&dev->power.lock) __acquires(&dev->power.lock)
{
    int(*callback)(struct device *);
    struct device *parent = NULL;
    int retval;

    dev_dbg(dev, "%s flags 0x%x\n", __func__, rpmflags);

repeat:
    //检查是否能执行 suspend 操作,主要是检查设备是否有 error 状态,操作是否 enable,
    //使用计数,子设备的状态(通常只有子设备都 suspend,父设备才能 suspend)等
    retval = rpm_check_suspend_allowed(dev);

    if(retval < 0)
        ;        /* Conditions are wrong. */
    /* Synchronous suspends are not allowed in the RPM_RESUMING state. */
    //已经在 resume 的过程中,并且要求 suspend 操作马上执行则报错,保证操作的同步
    else if(dev->power.runtime_status == RPM_RESUMING &&
        !(rpmflags & RPM_ASYNC))
        retval = -EAGAIN;
    //只有 retval 为 0 表示没有问题
    if(retval)
        goto out;

    /* If the autosuspend_delay time hasn't expired yet, reschedule. */
    //进行自动 suspend 操作,但是定时器还没有到时则进行必要的 reschedule 操作
    if((rpmflags & RPM_AUTO)
        && dev->power.runtime_status != RPM_SUSPENDING) {
        unsigned long expires = pm_runtime_autosuspend_expiration(dev);

        if(expires != 0) {
            /* Pending requests need to be canceled. */
```

```
                dev -> power. request = RPM_REQ_NONE;
                if( ! (dev -> power. timer_expires && time_before_eq(
                    dev -> power. timer_expires, expires))) {
                    //修改定时器
                    dev -> power. timer_expires = expires;
                    mod_timer( &dev -> power. suspend_timer, expires);
                }
                dev -> power. timer_autosuspends = 1;
                goto out;
            }
        }

        /* Other scheduled or pending requests need to be canceled. */
        //取消定时器并且直接修改设备的 runtime pm 的请求为 none,这样可以
        //在真正执行时直接返回,相当于撤销了 pending 请求.
        pm_runtime_cancel_pending( dev);

        //已经在执行 suspend 操作中,则要挂起,等待 suspend 操作完成
        if( dev -> power. runtime_status == RPM_SUSPENDING) {
            DEFINE_WAIT( wait);

            //如果是异步或者不等待的方式执行,都返回运行中的错误
            //异步操作返回该错误是由于本身已经在异步执行了
            if( rpmflags &( RPM_ASYNC | RPM_NOWAIT)) {
                retval = -EINPROGRESS;
                goto out;
            }

            /* Wait for the other suspend running in parallel with us. */
            //等待异步 suspend 的完成
            for( ;;) {
            prepare_to_wait( &dev -> power. wait_queue, &wait, TASK_UNINTERRUPTIBLE);
                //如果状态已经不是 suspending 了,表示已经完成退出循环
                if( dev -> power. runtime_status != RPM_SUSPENDING)
                        break;
                //调度之前释放锁
                spin_unlock_irq( &dev -> power. lock);

                schedule();
                //返回获得锁
                spin_lock_irq( &dev -> power. lock);
            }
```

```
        //结束等待则需要重新执行该流程来保证同步
        finish_wait(&dev->power.wait_queue, &wait);
        goto repeat;
    }

    //可以开始执行 suspend 操作了
    //先设置延迟 resume 状态
    dev->power.deferred_resume = false;
    //设置不执行操作,则直接设置状态
    if(dev->power.no_callbacks)
        goto no_callback;    /* Assume success. */

    /* Carry out an asynchronous or a synchronous suspend. */
    //如果是进行异步操作,则加入到 pm_wq 工作队列中执行
    if(rpmflags & RPM_ASYNC) {
  dev->power.request = (rpmflags & RPM_AUTO) ? RPM_REQ_AUTOSUSPEND:RPM_REQ_SUS-
PEND;
    if(! dev->power.request_pending) {
            dev->power.request_pending = true;
            queue_work(pm_wq, &dev->power.work);
        }
        //加入队列就可以退出
        goto out;
    }

    //设置 suspending 状态
    __update_runtime_status(dev, RPM_SUSPENDING);

    //根据设备的 runtime pm 的接口设置,选择操作接口
    if(dev->bus && dev->bus->pm && dev->bus->pm->runtime_suspend)
        callback = dev->bus->pm->runtime_suspend;
    else if(dev->type && dev->type->pm && dev->type->pm->runtime_suspend)
        callback = dev->type->pm->runtime_suspend;
    else if(dev->class && dev->class->pm)
        callback = dev->class->pm->runtime_suspend;
    else
        callback = NULL;
    //执行 runtime_suspend 操作
    retval = rpm_callback(callback, dev);
    if(retval) {
        //没有 suspend 成功,则恢复 active 状态
        __update_runtime_status(dev, RPM_ACTIVE);
```

```
            dev -> power.deferred_resume = 0;
            //如果是重新执行或者 busy 错误则清除错误值,否则取消 pending 的操作
            if(retval == -EAGAIN || retval == -EBUSY)
                dev -> power.runtime_error = 0;
            else
                pm_runtime_cancel_pending(dev);
        } else {
no_callback:
            //这里则 suspend 操作成功
            __update_runtime_status(dev, RPM_SUSPENDED);
            pm_runtime_deactivate_timer(dev);

            if(dev -> parent) {
                //对于父设备的 active 子设备数目做减操作
                parent = dev -> parent;
                atomic_add_unless(&parent -> power.child_count, -1, 0);
            }
        }
        //suspend 操作完成则唤醒其他等待的任务
        wake_up_all(&dev -> power.wait_queue);

        //如果延迟 resume 操作,这时可以执行了
        if(dev -> power.deferred_resume) {
            //执行 resume 操作
            rpm_resume(dev, 0);
            //suspend 操作没成功应该再操作
            retval = -EAGAIN;
            goto out;
        }

        //这里如果有父设备,在父设备考虑子设备状态时,考虑进入 idle
        if(parent && ! parent -> power.ignore_children) {
            //操作之前释放锁
            spin_unlock_irq(&dev -> power.lock);
            //异步执行父设备 idle 操作
            pm_request_idle(parent);
            //执行完获得锁
            spin_lock_irq(&dev -> power.lock);
        }

out:
```

```
        dev_dbg(dev, "%s returns %d\n", __func__, retval);

        return retval;
    }
```

从分析中可见，runtime pm 的操作很好地考虑了多上下文执行以及多种操作方式的情况，保证对于设备操作的同步。

除此之外系统提供了一系列的操作接口来帮助驱动进行 runtime pm 操作，下面是一些接口的例子：

```
//执行设备所在总线或者 class 应执行的 runtime_idle 操作
int pm_runtime_idle(struct device *dev);
//执行设备所在总线或者 class 应执行的 runtime_suspend 操作
int pm_runtime_suspend(struct device *dev);
//与 pm_runtime_suspend()相同,会考虑自动休眠延迟的时间
int pm_runtime_autosuspend(struct device *dev);
//执行设备所在总线或者 class 应执行的 runtime_resume 操作
int pm_runtime_resume(struct device *dev);
//提交执行设备所在总线或者 class 应执行的 runtime_idle 操作,由 pm_wq work 执行
int pm_request_idle(struct device *dev);
//提交执行设备所在总线或者 class 应执行的 runtime_resume 操作,由 pm_wq work 执行
int pm_request_resume(struct device *dev);
//递增设备的使用计数
void pm_runtime_get_noresume(struct device *dev);
//递增设备的使用计数,运行 pm_request_resume
int pm_runtime_get(struct device *dev);
//递增设备的使用计数,运行 pm_runtime_resume
int pm_runtime_get_sync(struct device *dev);
//递减设备的使用计数
void pm_runtime_put_noidle(struct device *dev);
//设备的使用计数减 1,如果结果是 0,则运行 pm_request_idle
int pm_runtime_put(struct device *dev);
//设备的使用计数减 1,如果结果是 0,则运行 pm_runtime_idle
int pm_runtime_put_sync(struct device *dev);
//设备的使用计数减 1,如果结果是 0,则运行 pm_runtime_suspend
int pm_runtime_put_sync_suspend(struct device *dev);

//使能 runtime pm 的辅助函数,允许执行设备所在总线或者 class 应执行的接口函数
void pm_runtime_enable(struct device *dev);
//防止新的执行请求,确保设备的所有等待中的操作已完成或取消
int pm_runtime_disable(struct device *dev);
```

具体设备的 runtime pm 的操作接口定义如下：

```
struct dev_pm_ops {
    ...
    int( * runtime_suspend) (struct device  * dev);
    int( * runtime_resume) (struct device  * dev);
    int( * runtime_idle) (struct device  * dev);
    ...
};
```

与 SLM 的操作接口类似都在 dev_pm_ops 中进行定义，会有设备及各种总线以及 class 进行实现，实际的操作也是要按照层次执行。

下面列举了一个设备驱动中如何使用 runtime pm 的简单例子。

probe 阶段时流程通常如下：

```
pm_runtime_enable
执行驱动自己的 probe 配置硬件
pm_runtime_suspend
```

需要进行工作时操作流程如下：

```
pm_runtime_get
具体工作
pm_runtime_put
```

没有具体的工作时可以直接通过 pm_runtime_suspend 来使得设备进入 suspend 状态。具体的操作由驱动根据实际情况调用 runtime pm 提供的各种接口来完成。

设备在电源管理方面除了 suspend 和 resume 的功能外，还有就是能够唤醒系统的能力。在唤醒系统的能力方面 device 中的 struct dev_pm_info 有 can_wakeup 属性表示 wakeup 的能力和 wakeup 属性表示唤醒源的信息等，通过这些属性来标记设备 wakup 的能力以及状态。另外提供了和 wakup 相关的接口如下：

```
extern struct wakeup_source  * wakeup_source_create(const char  * name);
extern void wakeup_source_destroy(struct wakeup_source  * ws);
extern void wakeup_source_add(struct wakeup_source  * ws);
extern void wakeup_source_remove(struct wakeup_source  * ws);
extern struct wakeup_source  * wakeup_source_register(const char  * name);
extern void wakeup_source_unregister(struct wakeup_source  * ws);
extern int device_wakeup_enable(struct device  * dev);
extern int device_wakeup_disable(struct device  * dev);
extern int device_init_wakeup(struct device  * dev, bool val);
extern int device_set_wakeup_enable(struct device  * dev, bool enable);
extern void __pm_stay_awake(struct wakeup_source  * ws);
extern void pm_stay_awake(struct device  * dev);
extern void __pm_relax(struct wakeup_source  * ws);
```

```
extern void pm_relax(struct device *dev);
extern void __pm_wakeup_event(struct wakeup_source *ws, unsigned int msec);
extern void pm_wakeup_event(struct device *dev, unsigned int msec);
```

这里的接口主要满足对设备在一定时间内唤醒系统的相关需求，具体的 wakeup 源是定时器，驱动可以根据需要进行调用。另外在设备 suspend 的时候还可以通过 device_may_wakeup 来检查设备是否可能唤醒系统而进行必要的唤醒设置（主要通过对 set_irq_wake 的封装接口 set_irq_wake 来实现）。

至此对主要的电源管理功能都进行了介绍和分析。

5.6 内核提供的同步操作、异步事件与单独执行实体的服务

5.6.1 同步操作服务

1. 等待队列 wait queue

用户对数据的访问有同步和异步两种方式，用户同步的访问在内核中通常并不是直接执行的，而是需要在另外的上下文执行。这就需要内核针对这种同步操作提供相应的服务，通常的方式就是用户的上下文进行等待，等到操作完毕再进行唤醒，这种等待和唤醒都需要在驱动内部执行，只是驱动在不同的上下文执行罢了。内核针对这种情况提供的同步操作服务就是等待队列 wait_queue。

等待队列框图如图 5-35 所示。

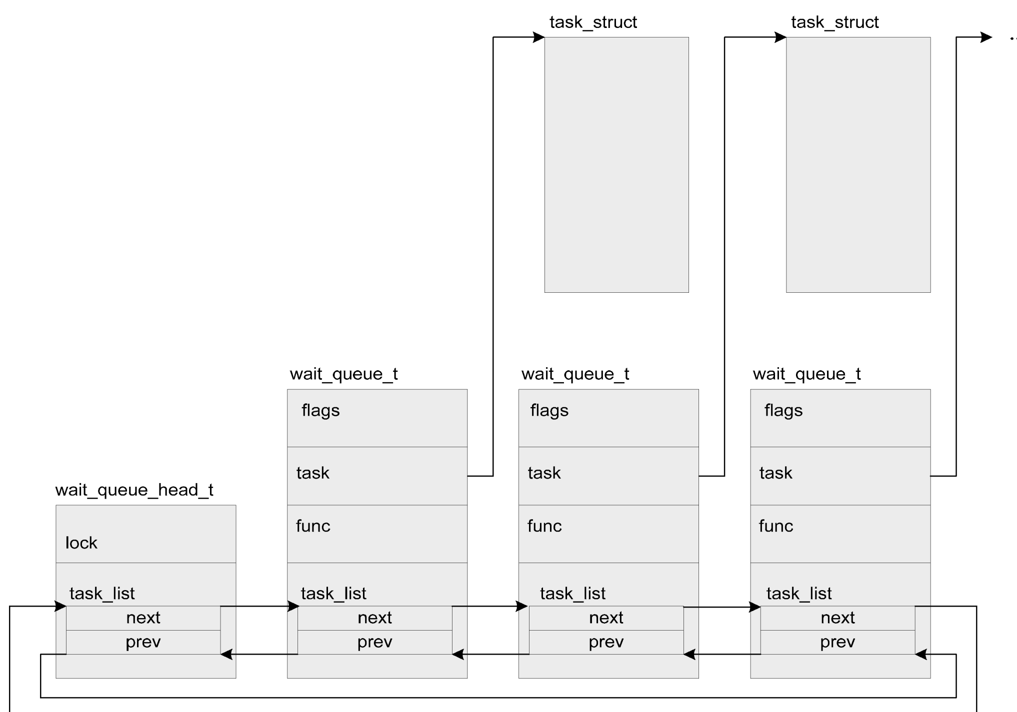


图 5-35 等待队列框图

从图 5-35 可见，wait_queue_head_t 为等待队列的头。如果需要等待相应的 task 则会通过 wait_queue_t 加入到等待队列中。相应的操作围绕着 wait_queue_head_t、wait_queue_t 和 task 来完成。下面介绍主要的接口。

定义和初始化等待队列头：

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
DECLARE_WAIT_QUEUE_HEAD(my_queue); 宏定义实现
```

定义并初始化等待队列项：

```
DECLARE_WAITQUEUE(name, tsk);
```

添加/移除等待队列项：

```
void fastcall add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
void fastcall remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
```

等待事件：

```
wait_event(wq, condition);不可中断的等待
wait_event_interruptible(wq, condition);可中断的等待
wait_event_timeout(wq, condition, timeout);带超时返回的等待
wait_event_interruptible_timeout(wq, condition, timeout);可中断并超时返回的等待
```

唤醒队列：

```
wake_up(wait_queue_head_t *q) ; 唤醒 q 上所有等待的进程
wake_up_interruptible(wait_queue_head_t *q); 只唤醒 q 上执行可中断休眠的进程
```

下面是驱动中将任务加入等待队列的基本流程：

```
//定义等待队列
DECLARE_WAITQUEUE(wait, current);
...
//增加等待队列进入读等待队列头,队列头在设备管理实体中定义
add_wait_queue(&dev->wait_head, &wait);
…
//设置进程状态为睡眠
__set_current_state(TASK_INTERRUPTIBLE);
…
//调度其他进程执行
schedule();
```

其中会包括必要的数据保护操作以及设备状态的检查。这样希望同步操作的任务就可以等待相关操作的完成了。在驱动操作完成的回调中，通过等待队列的唤醒队列的接口，唤醒相关的上下文继续执行，使得 schedule 返回，继续进行后续的操作，这样就完成同步服务了。

2. 完成量 completion

除了针对应用 task 的同步操作服务外，内核经常会有任务需要同步，比如初始化时某个任务需要等待另外任务的一个操作完成等。这只是两个任务之间简单的同步，使用等待队列显得大材小用了，因此内核提供完成量 completion 来进行这种类型的操作。相关接口如下：

```
init_completion( struct completion * ); 初始化指定的动态创建的完成变量
wait_for_completion( struct completion * ); 等待指定的完成变量接收信号
complete( struct completion * ); 发信号唤醒任何等待任务
```

从接口可见，完成量十分简洁，非常适合两个任务之间的同步服务。

5.6.2 异步事件

1. 异步 IO AIO

除了同步数据读写之外，内核还提供了异步读写功能，对于 VFS 文件相关的接口是 aio_read 和 aio_write 等。AIO 帮助用户程序提高性能以提高整体 CPU 和 IO 设备的利用率，特别是高 IO 负载的效率。在服务型应用中比较广泛的应用，比如各种代理服务器，数据库，流服务器等等。

AIO 可以一次性发出大量的 read/write 请求，请求完成后统一返回。这样在用户程序的角度减少了因同步操作产生的负载，相当于应用层的数据批处理功能。

Linux 内核为 AIO 提供了系统调用来完成异步 IO 的功能，相关调用如下：

- io_setup( )；为当前进程建立异步 IO 上下文。
- io_submit( )；提交一个或者多个异步 IO 操作。
- io_getevents( )；获得完成的异步 IO 的状态。
- io_cancel( )；取消某个 IO 操作。
- io_destroy( )；销毁异步 IO 上下文。

下面提供一个简单的例子，便于了解如何使用异步 IO。具体如下：

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <libaio.h>
```

```
int main( )
{
    io_context_t ctx;
    struct iocb  * iocb;
    struct io_event events[10];

    char *  buf;
    int fd;
    unsigned nr_events = 10;
    struct timespec timeout = {1, 100};

    //初始化异步 IO 上下文管理结构
    memset(&ctx, 0, sizeof(ctx));
    //创建异步 IO 上下文
    io_setup(nr_events, &ctx);

    //目前要指定 O_DIRECT,这样 io_submit 操作才能使用异步 IO
    int fd = open("./test.txt", O_DIRECT | O_WRONLY, S_IRWXU | S_IRWXG | S_IROTH);

    //分配的空间要求对齐
    posix_memalign((void **)&buf, sysconf(_SC_PAGESIZE), sysconf(_SC_PAGESIZE));
    //在相应空间写入内容
    strcpy(buf, "test");

    //分配异步 IO 操作结构
    iocb = (struct iocb  * )malloc(sizeof(struct iocb));
    memset(iocb, 0, sizeof(struct iocb));

    //设定异步 IO 操作
    iocb[0].data              = buf;
    iocb[0].aio_lio_opcode    = IO_CMD_PWRITE;
    iocb[0].aio_reqprio       = 0;
    iocb[0].aio_fildes        = fd;

    iocb[0].u.c.buf           = buf;
    // 这个值必须按 512 B 对齐
    iocb[0].u.c.nbytes        = page_size;
    iocb[0].u.c.offset        = 0;

    //提交异步 IO 操作,这里是异步写磁盘
    io_submit(ctx, 1, &iocbpp);
```

```
        //检查写磁盘情况,类似于 epoll_wait 或 select
        io_getevents( ctx , 1 , 10 , events , &timeout) ;

        //关文件
        close( fd) ;
        //销毁异步 IO 上下文
        io_destroy( ctx) ;
        return 0;
    }
```

从应用代码可见，异步 IO 更适合直接对块设备进行操作。通过进行一次大量的异步读写来减少同步开销，从而提高效率，这种场景更适合使用块设备的服务，如数据库、流服务等。

异步 IO 的内核实现框架如图 5-36 所示。

图 5-36　异步 IO 内核实现框架

从图 5-36 可见，每个异步 IO 的上下文都包含真正执行 IO 操作的实体 work（在 worker 线程上执行），而 IO 操作的完成情况通过 ring buffer 实现内核与用户之间交互。

相应的执行实体初始化是在分配 IO 上下文的函数 ioctx_alloc 来进行的，具体如下：

```
INIT_DELAYED_WORK(&ctx->wq, aio_kick_handler);
```

在需要执行 IO 操作时，会调度 work 执行 aio_kick_handler。其中，调用 aio_run_iocb 对每个 kiocb 进行操作，调用 ki_retry 执行具体的操作。而 kiocb 在创建时会根据具体的 IO 操作通过 aio_setup_iocb 分配合适的操作接口给 ki_retry。读写操作通常设置为 aio_rw_vect_retry，下面来分析一下 aio_rw_vect_retry 的实现：

```
static ssize_t aio_rw_vect_retry(struct kiocb *iocb)
{
    struct file *file = iocb->ki_filp;
    struct address_space *mapping = file->f_mapping;
    struct inode *inode = mapping->host;
    ssize_t(*rw_op)(struct kiocb *, const struct iovec *,
            unsigned long, loff_t);
    ssize_t ret = 0;
    unsigned short opcode;

    //根据操作,从文件操作接口中获得正确的操作函数
    if((iocb->ki_opcode == IOCB_CMD_PREADV) ||
        (iocb->ki_opcode == IOCB_CMD_PREAD)) {
        rw_op = file->f_op->aio_read;
        opcode = IOCB_CMD_PREADV;
    } else {
        rw_op = file->f_op->aio_write;
        opcode = IOCB_CMD_PWRITEV;
    }

    /* This matches the pread()/pwrite() logic */
    if(iocb->ki_pos < 0)
        return -EINVAL;

    //遍历对请求的所有 IO 进行操作
    do {
        //执行文件的异步接口
        ret = rw_op(iocb, &iocb->ki_iovec[iocb->ki_cur_seg],
                iocb->ki_nr_segs - iocb->ki_cur_seg,
                iocb->ki_pos);
        if(ret > 0)
            aio_advance_iovec(iocb, ret);

    /* retry all partial writes.  retry partial reads as long as its a
     * regular file.  */
```

```
    } while(ret > 0 && iocb -> ki_left > 0 &&
        (opcode == IOCB_CMD_PWRITEV ||
        (! S_ISFIFO(inode -> i_mode) && ! S_ISSOCK(inode -> i_mode))));

    /* This means we must have transferred all that we could */
    /* No need to retry anymore */
    //一切正常则返回操作数目
    if((ret == 0) || (iocb -> ki_left == 0))
        ret = iocb -> ki_nbytes - iocb -> ki_left;

    /* If we managed to write some out we return that, rather than
     * the eventual error. */
    if(opcode == IOCB_CMD_PWRITEV
        && ret < 0 && ret != -EIOCBQUEUED && ret != -EIOCBRETRY
        && iocb -> ki_nbytes - iocb -> ki_left)
        ret = iocb -> ki_nbytes - iocb -> ki_left;

    return ret;
}
```

从代码中可见，其主要是通过文件的异步接口来执行所有的异步请求，而相应的异步接口通常由具体的文件系统提供，如 ext2 文件系统的 ext2_file_operations 中就包含各种异步操作接口。具体的文件系统后续的操作在块设备层已经进行了介绍。可见 Linux 内核已经为异步 IO 提供了完整的上层框架，对于具体的操作则是通过 VFS 中的异步接口完成的。这样将具体的文件 IO 操作转移到异步 IO 框架中执行，用户层则不需要同步等待事件的完成，从而在需要进行大量操作时整体的性能要高很多。

2. 通知链 notifier

Linux 内核的各个子系统功能上都是相互独立的，但是各个子系统之间同样需要交互，例如某个子系统可能对其他子系统产生的事件感兴趣。如何让各个子系统之间能够方便地通知相关的事件并进行合适的操作，这属于横切的异步功能。Linux 内核为了满足该需求设计了通知链的机制。通知链是为内核各个模块异步交互服务的，只能够在内核的子系统之间使用，而不能够用于内核与用户空间之间进行事件的交互。

通知链表是一个函数链表，链表上的每一个节点都注册了一个函数。当某个事件发生时，链表上所有节点对应的函数就会被执行。所以通知链表有一个通知方与很多接收方。在通知这个事件时所运行的函数由接收方决定，就是接收事件方注册了回调函数，在发生某个事件时回调函数就得到执行。系统运行时通知链的状态如图 5-37 所示。

从图 5-37 可见，其中包含两个主要的实体，一个是 head，另一个是 block。head 是由事件通知者提供，而 block 是由事件接收者向事件通知者注册的。

Linux 内核提供与通知链相关的结构体如下：

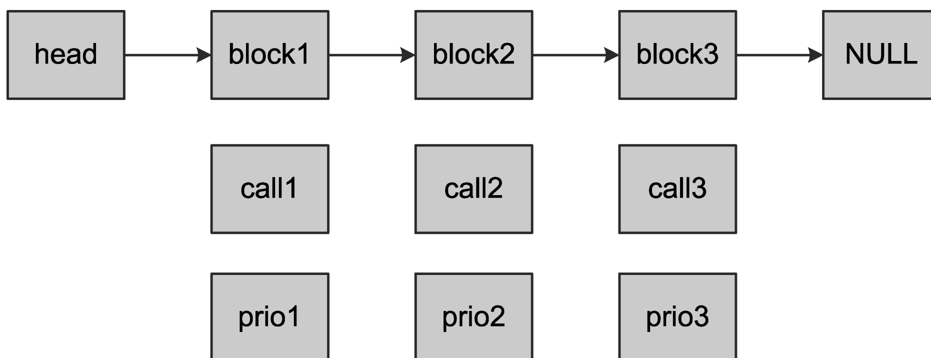


图 5-37　通知链运行时状态图

```
struct notifier_block {
    int( * notifier_call)(struct notifier_block * , unsigned long, void * );
    struct notifier_block __rcu * next;
    int priority;
};

struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block __rcu * head;
};

struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block __rcu * head;
};

struct raw_notifier_head {
    struct notifier_block __rcu * head;
};

struct srcu_notifier_head {
    struct mutex mutex;
    struct srcu_struct srcu;
    struct notifier_block __rcu * head;
};
```

从中可见，有一个与 block 相关的结构 notifier_block，有四个 head 相关的结构 xxxx_notifier_head。为什么有四种事件通知者呢？这是由于内核事件本身可能在不同的上下文中发生，通常在事件发生时就要进行事件通知，而对通知的回调操作会有限制，Linux 内核就直接将这些限制通过通知者的结构实现，这样依据不同的情况产生了四种 head，细节如下：

- atomic_notifier_head 通知链 block 的回调函数（当事件发生时要执行的函数）只能在

中断上下文中运行，不允许阻塞。

- blocking_notifier_head 通知链元素的回调函数在进程上下文中运行，允许阻塞。
- raw_notifier_head 对通知链元素的回调函数没有任何限制，所有锁和保护机制都由调用者维护。
- srcu_notifier_head 可阻塞通知链的变种，使用 SRCU （sleepable read - copy update）来替代 rw - semaphores 进行通知链的保护。

根据不同类型的通知链，内核提供了不同的注册函数，具体如下：

```
extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
        struct notifier_block *nb);
extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *nb);
extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *nb);
extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

相应的内核还提供了事件发布函数，由通知者调用来进行事件发布并对通知链中注册的接收者操作进行调用，细节如下：

```
extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
        unsigned long val, void *v);
extern int __atomic_notifier_call_chain(struct atomic_notifier_head *nh,
        unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v);
extern int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int raw_notifier_call_chain(struct raw_notifier_head *nh,
        unsigned long val, void *v);
extern int __raw_notifier_call_chain(struct raw_notifier_head *nh,
        unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int srcu_notifier_call_chain(struct srcu_notifier_head *nh,
        unsigned long val, void *v);
extern int __srcu_notifier_call_chain(struct srcu_notifier_head *nh,
        unsigned long val, void *v, int nr_to_call, int *nr_calls);
```

Linux 内核还提供了 head 的定义宏，细节如下：

```
#define ATOMIC_NOTIFIER_HEAD(name) \
    struct atomic_notifier_head name = \
        ATOMIC_NOTIFIER_INIT(name)
#define BLOCKING_NOTIFIER_HEAD(name) \
```

```
        struct blocking_notifier_head name = \
            BLOCKING_NOTIFIER_INIT(name)
    #define RAW_NOTIFIER_HEAD(name) \
        struct raw_notifier_head name = \
            RAW_NOTIFIER_INIT(name)
```

具体的事件是由通知者模块进行定义的，相应的接收者也需要清楚具体的事件，并在回调中根据事件进行正确的操作。对接收者来说，重要的是将 block 注册加入正确的通知链中，要做到这一点需要知道具体的 head。这相当于要知道模块的细节，提高了模块间的耦合度，更好的方法是事件的发布者所在的模块提供一个接口函数用于注册 block 信息，这样接收者就不需要关心事件发布模块的细节，降低了通知链间各模块的耦合度。Linux 内核也是通过该方法实现的。

内核运行过程中会有各种各样的异步事件发生，而通过通知链技术使得系统中各个模块的交互变得简洁，事件的发布者只要将事件发布即可，可以不用关心接收者的细节，接收者则只关心事件而不用关心发布者的实现，这样的设计简单、清晰并且耦合度也是很低的。

5.6.3 单独执行实体服务

1. 工作队列 work queue

Linux 内核中很多模块会有一些延时操作的需求，由于内核中各种上下文能进行的操作不同，所以延时操作最简单的办法就是能够在单独的进程上下文中执行，这样可以进行各种复杂的操作而相对限制最少。如果让内核中各个模块的延时操作都要创建进程，则系统运行时会占用大量的资源，同时也增加了系统调度的开销。如何做才是最好的呢？对延时操作来说其抽象的概念是操作，而操作的软件化抽象就是函数。如果系统可以根据当前系统的运行情况动态地创建执行实体，而当模块有延时操作需要时直接将这些操作本身加入到系统提供的进程上执行，这样既满足了各个模块的功能需求，又可以使系统资源占用最少，兼顾了功能和性能两个方面。Linux 内核提供的工作队列就实现了该功能，对各个模块来说，其需要系统进行的延时操作就是一个一个的“工作”。

Linux 内核运行时工作队列状态如图 5-38 所示。

从图 5-38 可见，Linux 内核根据需要在不同的处理器上创建执行实体线程，来执行具体的 work，这些 work 都是动态添加的。整个 work queue 的管理实体是 workqueue_struct，其中可以包含多个线程，分别执行相应的延时工作。

Linux 内核为系统提供了一些默认的 work queue，具体如下：

```
    extern struct workqueue_struct *system_wq;
    extern struct workqueue_struct *system_long_wq;
    extern struct workqueue_struct *system_nrt_wq;
    extern struct workqueue_struct *system_unbound_wq;
```

这些 work queue 会由不同的接口向其中添加工作，每个工作队列的执行特点也是不同的，system_wq 会有多个任务执行，但是希望每个 work 不要消耗太长时间；system_long_wq

类似于 system_wq，但是允许长时间的 work 执行；system_nrt_wq 确保 work 不会重入，并且不会在多个处理器上执行；system_unbound_wq 则不会将 work 固定在某个处理器上执行，并将尽快尽力执行。通常默认使用的是 system_wq，也可以根据需要通过 queue_work 将 work 加入到具体的 work queue 上来执行。

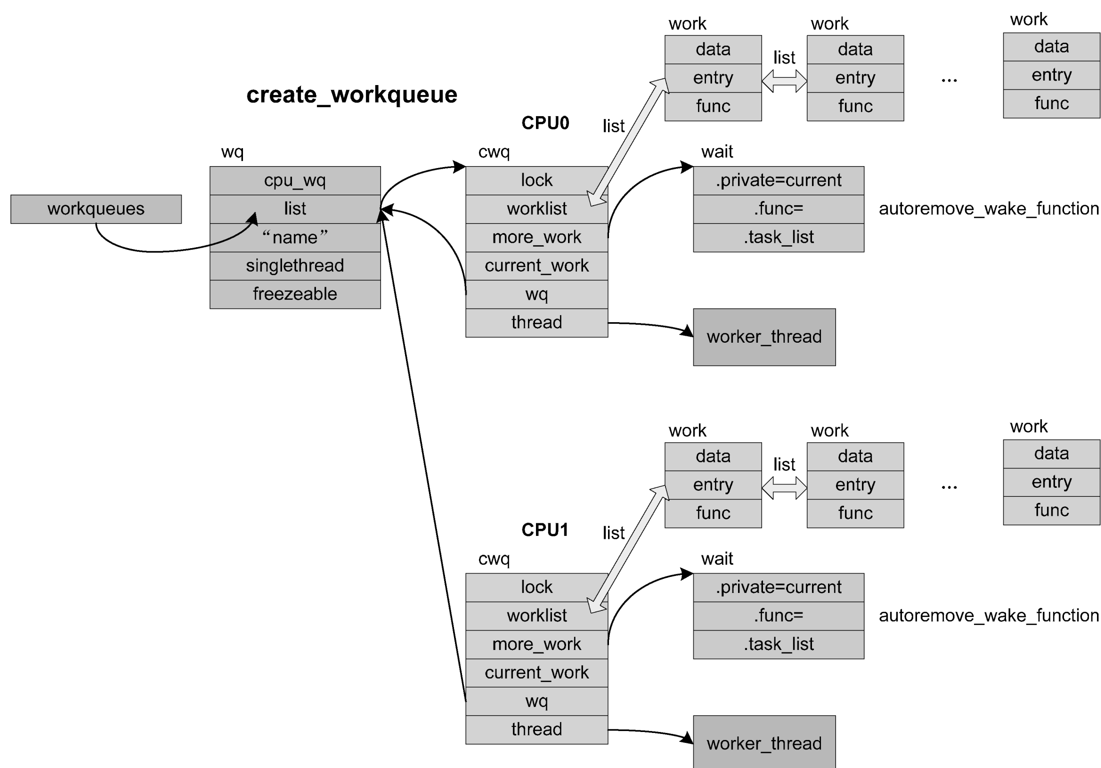


图 5-38　工作队列运行状态

由于系统已经提供了各种类型的工作队列，对模块开发来说，延时操作最重要的就是 work 了，系统对 work 管理的结构是 work_struct，细节如下：

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

结构很简单，其中对模块最重要的是操作接口 work_func_t，具体定义如下：

```
typedef void( * work_func_t)(struct work_struct * work);
```

从中可见传入的参数并不是模块相关的，那么模块如何获得其管理实体的信息呢？方法就是将 work_struct 做成静态的形式并嵌入模块管理实体中，这样通过 container_of 就可以获得相关的管理实体。为此工作队列框架提供了相应的初始化宏：

```
#define INIT_WORK(_work, _func)\
    do {\
        __INIT_WORK((_work),(_func), 0);\
    } while(0)
```

工作队列框架还提供了一系列接口函数方便各个模块使用，接口说明如下：

- create_workqueue：用于创建一个 workqueue 队列，为系统中的每个 CPU 都创建一个内核线程。
- create_singlethread_workqueue：用于创建 workqueue，只创建一个内核线程。
- destroy_workqueue：释放 workqueue 队列。
- schedule_work：调度执行一个具体的任务 work_struct，执行的任务将会被挂入 Linux 系统提供的 system_wq：名字是 events。
- schedule_delayed_work：延迟一定时间去执行一个具体的任务，功能与 schedule_work 类似，多了一个延迟时间。
- queue_work：调度执行一个指定 workqueue 中的任务。
- queue_delayed_work：延迟调度执行一个指定 workqueue 中的任务，功能与 queue_work 类似，输入参数多了一个 delay。
- queue_work_on：调度执行一个指定 workqueue 中的任务，并在指定的处理器上执行。

对于通常的模块来说不需要创建工作队列，而是通过 INIT_WORK 来初始化静态的 work_struct，在需要调度 work 执行时（如中断处理 ISR 中），通过相应的接口进行调度即可。

2. 内核线程 kernel thread

工作队列并不能满足所有的需求，另外工作队列本身也是需要一个真正的执行实体的承载，在内核中这个真正的执行实体就是内核线程 kernel thread。在执行 ps -aux 命令时会看到如下信息：

```
root    2   0.0  0.0   0   0 ?   S   11:57   0:00 [kthreadd]
root    3   0.0  0.0   0   0 ?   S   11:57   0:00 [ksoftirqd/0]
root    6   0.0  0.0   0   0 ?   S   11:57   0:00 [migration/0]
root    7   0.0  0.0   0   0 ?   S   11:57   0:00 [watchdog/0]
root    8   0.0  0.0   0   0 ?   S   11:57   0:00 [migration/1]
root    10  0.0  0.0   0   0 ?   S   11:57   0:00 [ksoftirqd/1]
root    11  0.0  0.0   0   0 ?   S   11:57   0:00 [watchdog/1]
```

这些都是内核线程，之前在介绍中断时，介绍了中断处理也提供了在内核线程上执行操作的接口。

内核线程作为独立的调度实体，对于内核模块来说控制能力更强，所以很多复杂的功能都需要通过内核线程来执行。

下面对内核线程的主要接口进行说明：

```
struct task_struct * kthread_create(int( * threadfn)(void * data),
                                    void * data,
                                    const char namefmt[], ...)
    __attribute__((format(printf, 3, 4)));
```

kthread_create 用以产生内核线程，可以在所有处理器上运作，产生后的内核线程会等待被 wake_up_process 唤醒或是被 kthread_stop 终止。

```
#define kthread_run(threadfn, data, namefmt, ...)    \
({    \
    struct task_struct * __k    \
        = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
    if(! IS_ERR(__k))    \
        wake_up_process(__k);    \
    __k;    \
})
```

kthread_run 主要用于产生并唤醒内核线程（开发者可以省去要呼叫 wake_up_process 的动作），其是基于 kthread_create，所以产生的内核线程也不限于在特定的处理器上执行。

```
void kthread_bind(struct task_struct * p, unsigned int cpu);
```

kthread_bind 用来将内核线程绑定到固定的 CPU 上，主要是通过设定 CPU allowed bitmask 来实现，所以在 SMP 的架构下，就可以指定给一个以上的处理器执行。

```
int kthread_stop(struct task_struct * k);
```

kthread_stop 用来暂停通过 kthread_create 产生的内核线程，并会等待内核线程结束，并传回函式 threadfn 的返回值。

通过这些接口可见，主要的内核线程都是通过 kthread_create 创建的，下面来看看其具体的细节。

```
struct task_struct * kthread_create(int( * threadfn)(void * data),
                                          void * data,
                                          const char namefmt[],
                                          ...)
{
    struct kthread_create_info create;

    //初始化创建信息
    create.threadfn = threadfn;
    create.data = data;
    init_completion(&create.done);

    spin_lock(&kthread_create_lock);
```

```
    //添加到创建列表中
    list_add_tail(&create.list, &kthread_create_list);
    spin_unlock(&kthread_create_lock);

    //唤醒 kthreadd 内核线程,通过创建参数创建内核线程
    wake_up_process(kthreadd_task);
    //等待创建完成
    wait_for_completion(&create.done);

    if(! IS_ERR(create.result)) {
        struct sched_param param = { .sched_priority = 0 };
        va_list args;

        va_start(args, namefmt);
        vsnprintf(create.result->comm, sizeof(create.result->comm),
                namefmt, args);
        va_end(args);
        /*
         * root may have changed our(kthreadd's) priority or CPU mask.
         * The kernel thread should not inherit these properties.
         */
        sched_setscheduler_nocheck(create.result, SCHED_NORMAL, &param);
        set_cpus_allowed_ptr(create.result, cpu_all_mask);
    }
    return create.result;
}
```

可见，创建内核线程的主要任务是由 kthreadd 内核线程来实现的，kthreadd 内核线程是系统的 2 号线程，主要的任务就是屏蔽体系结构相关的部分，为系统提供创建内核线程的服务，其与系统初始化相关的部分之前已经进行了介绍，其具体创建线程的操作是通过 kernel_thread 实现的，ARM 内的实现如下：

```
pid_t kernel_thread(int(*fn)(void *), void *arg, unsigned long flags)
{
    struct pt_regs regs;

    memset(&regs, 0, sizeof(regs));

    regs.ARM_r4 = (unsigned long)arg;
    regs.ARM_r5 = (unsigned long)fn;
    regs.ARM_r6 = (unsigned long)kernel_thread_exit;
    regs.ARM_r7 = SVC_MODE | PSR_ENDSTATE | PSR_ISETSTATE;
```

```
        regs.ARM_pc = (unsigned long)kernel_thread_helper;
        regs.ARM_cpsr = regs.ARM_r7 | PSR_I_BIT;

        return do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);
    }
```

可见，其主要是通过 do_fork 这一调度模块的接口创建的。

这样，内核线程的主要功能和接口就进行了介绍，内核线程主要提供执行实体的创建和管理服务，具体的业务细节要由具体的模块提供操作接口执行。

5.7 内核提供的数据保护一致性操作服务

5.7.1 数据保护一致性操作服务的需求

Linux 内核的执行会有各种各样复杂的情况，处理器技术的发展为系统的执行提供了更多的变数。多处理器、多流水、乱序执行都对系统的数据一致性有很大影响，在关键点的数据操作上，如果不能保证一致性，结果就会导致错误，甚至产生严重性的问题，这些问题对于驱动来说更是如此，如读取设备寄存器的状态值与实际的不同时，设备驱动的操作必然是错误的，所以从系统的角度，不同硬件模块操作就需要保证数据的一致性，对这些数据的操作进行保护。只有这样才能保证系统正确运行。

5.7.2 各种数据保护一致性操作简介

在内核提供的基本服务中已经简单介绍了一些锁的功能，这里对与驱动相关的数据一致性保护服务进行进一步介绍。

1. 顺序和屏障

驱动经常要涉及很多读写操作，设备的读写操作很多是要有顺序性的，对编译器和处理器来说并不能在之前就进行这种顺序性的假设，而是尽力优化代码，相应的驱动就需要保证这种顺序性，这是矛盾的，不过还好编译器和处理器都提供了相关功能来保证操作的顺序性。Linux 内核对编译器和处理器的相关指令和操作进行了封装，提供给其他模块使用，相关的操作就是形成一个屏障，屏障保证之前和之后的某些操作的顺序性，下面是相关的接口及说明：

- rmb()：阻止跨越屏障的读操作发生重排序，保证前后读操作的顺序性。
- read_barrier_depends()：阻止跨越屏障的具有数据依赖关系的读操作重排序。
- wmb()：阻止跨越屏障的写操作发生重排序。
- mb()：阻止跨越屏障的读和写操作重新排序。
- smp_rmb()：在 SMP 上提供 rmb()功能，在 UP 上提供 barrier()功能。
- smp_read_barrier_depends()：在 SMP 上提供 read_barrier_depends()功能，在 UP 上提供 barrier()功能。
- smp_wmb()：在 SMP 上提供 wmb()功能，在 UP 上提供 barrier()功能。

- smp_mb()：在 SMP 上提供 mb()功能，在 UP 上提供 barrier()功能。
- barrier()：阻止编译器跨越屏障对读或写操作进行优化。

2. 内核抢占

Linux 内核提供了内核抢占之后在一定程度上也会造成数据的不一致，比如说进程的切换造成数据访问的交叉执行，这样只在进程上下文的数据操作可以考虑通过内核抢占的接口来保证数据一致性，接口说明如下：

- preempt_disable()：增加抢占计数值，从而禁止内核抢占。
- preempt_enable()：减少抢占计算，并当该值降为 0 时检查和执行被挂起的需调度的任务。
- preempt_enable_no_resched()：激活内核抢占但不再检查任何被挂起的需调度的任务。
- preempt_count()：返回抢占计数。

这里的 preempt_disable()和 preempt_enable()是可以嵌套调用的，disable 和 enable 的次数最终应该是一样的。

另外在中断上下文和进程上下文中需要的数据一致性保护可以通过中断开关解决，多处理器操作的数据一致性保护可以通过自旋锁解决，在实际的应用中自旋锁的接口提供了相关中断的操作，所以通常都直接使用正确的自旋锁接口进行相关的保护操作。还有原子操作和互斥锁也是数据一致性保护操作，原子操作主要针对单变量的数据，没有上下文的限制，而互斥锁通常是在多个进程上下文中进行数据一致性的保护。这些接口都在内核基本服务中有简单介绍，这里就不重复说明了。

还有各种 RCU（主要差别在如何进行回写上）、读写锁等，通常都是针对网络协议栈、文件系统等有一定读写特点的数据一致性保护操作，在驱动中使用的情况并不多，就不进行详述了。

总之对数据一致性保护操作，是要在正确的前提下，尽量根据数据操作的上下文以及特点选择合适的接口进行操作，在完成功能的基础上尽力提升性能。

5.8 小结

本章主要介绍和分析了驱动框架的实现，以及 Linux 内核为设备驱动提供的基础服务。这些内容都是了解设备驱动的基础，只有熟悉了它们才能更好更清晰地了解设备驱动的机理。通过驱动框架合理的使用内核提供的这些基础服务才能更好地实现设备驱动。

在分析和介绍的过程中从需求出发，尽量以层次和功能的方式进行总结，尽力做到知道为什么这么做，这样在使用时才会有的放矢。

第 6 章　设备驱动之功能型驱动

本章主要是对功能类设备的驱动框架进行介绍，其中还会对应用层如何使用相应的设备以及 Android 框架的适配进行介绍。

6.1　输入设备（input）

6.1.1　输入设备需求

输入设备是提供给用户输入数据和信息的设备，是用户与设备沟通的桥梁。键盘、鼠标、摄像头、光笔、手写输入板、游戏杆、语音输入装置等都属于输入设备。Linux 内核提供框架来对各种物理的输入设备进行支持，摄像头、语音输入是属于音视频相关的功能，这两类设备的数据包含大量的信息，它们有专门的框架进行支持。而其他如键盘、鼠标等输入信息偏向控制信息，在 Linux 内核中统一由输入设备 Input 框架来支持。

从物理设备的角度来说，输入设备 Input 框架需要能够支持各种不同类型的输入设备。而从上层用户的角度，用户希望能够屏蔽不同设备的差异，有统一的接口和操作方法以及能够接收标准的数据，这也是需要输入设备 Input 框架解决的问题。总的来说，输入设备框架需要在多样的物理设备和统一的用户标准之间进行转换和平衡。

6.1.2　输入设备框架解析

1. 框架总体设计

从输入设备框架的需求可知，内核的整个输入流程应该分为下层的各种输入设备和上层与用户的控制信息交互两个方面。所以输入设备框架重要的工作应该是在管理下层的各种输入设备和上层的控制信息交互处理实体，并为用户层提供统一的服务，内部框架还需要能够在输入设备和用户的控制信息两者之间建立有效的通道，从而进行信息交互。

根据这些目标，输入设备框架的总体设计如图 6-1 所示。

从图 6-1 可见，输入设备系统框架的核心部分从底到上分为 input driver，input core 和 event handler 三个部分。详细介绍如下：

- input driver 主要负责下层物理输入设备的驱动及管理，每个具体的驱动都会对物理设备进行实际的操作。
- event handler 主要负责为应用层提供获得设备输入的信息，将输入事件以标准形式提供给应用层，并接受应用层的控制。
- Input core 主要负责对 Input driver 和 Event handler 进行管理，将二者进行匹配并在二者之间建立通信的通道。总的来说起到连接和桥梁的作用。

从图 6-1 可见，对为用户开放的设备文件，每个 event handler 会有不同的设备文件，需要注意的是在/dev/input/目录下的文件才是属于输入设备系统框架生成的，而 console 和 tty

Input输入子系统框架

/dev/input/event0
/dev/input/event1
……
/dev/input/ts0
/dev/input/ts1
……
/dev/input/js0
/dev/input/js1
……
/dev/input/mice
/dev/input/mouse0
/dev/input/mouse1
……
/dev/console
/dev/ttyn
……
Event handler
evdev.c
tsdev.c
joydev.c
mousedev.c
keyboard.c
Input core
driver/input/input.c
Input driver
s3c2410ts.c
usbmouse.c
usbkbd.c
……

图 6-1　输入设备 Input 系统框架总体设计

属于终端设备类，并不属于输入设备系统框架对用户层开放的设备。这也就是说，输入事件的接收者不是必须产生针对用户的输入，只要关心输入事件的模块就可以注册 event handler，这一点很重要，也为许多功能开放了监测输入设备输入进行控制的接口，可以在内核内部就根据用户的某些输入做特定的控制。

根据以上的三个部分的功能可见，如果设备文件是由输入设备框架创建的，则相应的应用层的接口只有 event handler，所有与应用层的交互都是通过 event handler 来完成。event handler 需要对应用层的使用者进行管理，其中 evdev handler 的管理模式如图 6-2 所示。

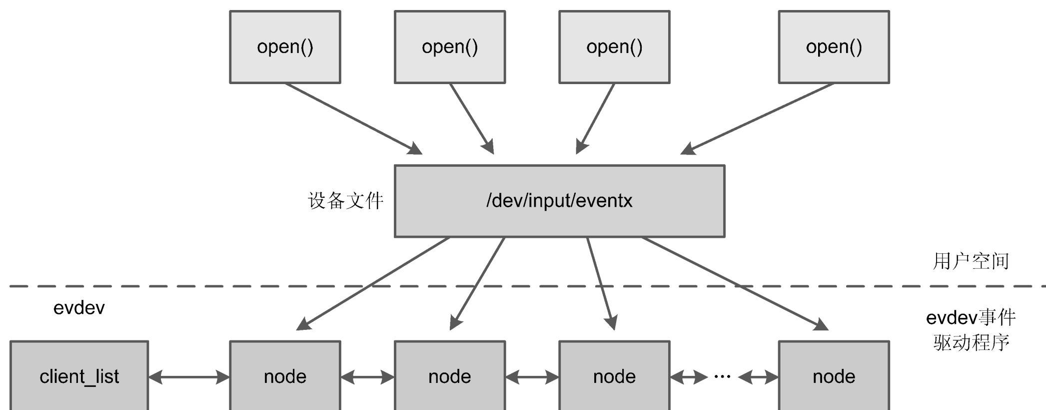


图 6-2　evdev handler 用户管理模式图

从图 6-2 可见，每个操作文件的进程都会对应一个 evdev_client，框架将对与之连接的输入设备上传的输入事件分别传入 evdev_client 进行缓冲与管理，保证用户进程的统一视角，evdev_client 则由 evdev 统一管理。struct evdev 在 evdev handler 中代表一个 eventX 文件，而 struct evdev 由 event handler 指向的表示连接通道管理实体 input_handle 中的 private 指示，这

样就完成了从抽象的 handler 管理到实体的 handler 的转换，进而使得系统从下层的设备层到上层的应用层的通路都是畅通的。

2. 主要管理实体及功能

按照子系统三部分的划分，管理实体大体也分为三种，首先来看看系统对于下层 input device 的管理实体，细节如下：

```
struct input_dev {
    //设备属性
    const char *name;
    const char *phys;
    const char *uniq;
    struct input_id id;

    //这里要支持各种的输入设备,所以要标记该输入设备具体的支持状况
    //能力表中设备将支持的功能相应的位标记为1
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

    unsigned int hint_events_per_packet;

    //设备特殊的 keycode 属性及操作接口
    unsigned int keycodemax;
    unsigned int keycodesize;
    void *keycode;

    int(*setkeycode)(struct input_dev *dev,
                unsigned int scancode, unsigned int keycode);
    int(*getkeycode)(struct input_dev *dev,
                unsigned int scancode, unsigned int *keycode);
    int(*setkeycode_new)(struct input_dev *dev,
                    const struct input_keymap_entry *ke,
                    unsigned int *old_keycode);
    int(*getkeycode_new)(struct input_dev *dev,
                    struct input_keymap_entry *ke);
```

```
struct ff_device  * ff;

//用于重复按键的操作及记录
unsigned int repeat_key;
struct timer_list timer;

int rep[REP_CNT];

struct input_mt_slot  * mt;
int mtsize;
int slot;

//用于绝对位置的设备记录信息
struct input_absinfo  * absinfo;

//设备保存相关操作信息
unsigned long key[BITS_TO_LONGS(KEY_CNT)];
unsigned long led[BITS_TO_LONGS(LED_CNT)];
unsigned long snd[BITS_TO_LONGS(SND_CNT)];
unsigned long sw[BITS_TO_LONGS(SW_CNT)];

//用于设备控制
int( * open)(struct input_dev  * dev);
void( * close)(struct input_dev  * dev);
int( * flush)(struct input_dev  * dev, struct file  * file);
//该接口用于某事件需要设备进行特殊操作,如按某些键需用设备层亮灯
int( * event)(struct input_dev  * dev, unsigned int type, unsigned int code, int value);

//对于设备只绑定单一的 event handler
struct input_handle __rcu  * grab;

spinlock_t event_lock;
struct mutex mutex;

unsigned int users;
bool going_away;

bool sync;

//设备模型的接口
struct device dev;
```

```
    //设备绑定多个 event handler,这里和 input_handle 绑定进而与 event handler 绑定
    struct list_head        h_list;
    //整体的设备管理
    struct list_head        node;
};
```

从 input_dev 中可见，它的属性兼容了各种输入设备，主要是在底层记录输入的数据和状态，便于进行一定的重复以及同时多事件（同时按键）的合并处理。

接下来是对于上层 event handler 的管理实体，细节如下：

```
struct input_handler {
    //特定 handler 的实例化管理信息
    void *private;

    //以下是事件相关的操作接口
    //处理输入事件的接口
    void(*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
    //过滤的接口
    bool(*filter)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
    //用于特殊的对是否支持设备的检查
    bool(*match)(struct input_handler *handler, struct input_dev *dev);
    //用于对 input device 和 event handler 的绑定,这里做绑定设备后的初始化
    int(*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device
    _id *id);
    //和设备脱离绑定
    void(*disconnect)(struct input_handle *handle);
    //一般用于设备绑定后的 handler start 工作,通常为 null
    void(*start)(struct input_handle *handle);

    //和上层文件的接口
    const struct file_operations *fops;
    //子设备号,只有会产生设备文件的 handler 才有该值
    int minor;
    const char *name;

    //要求设备匹配的能力信息,可以是强制匹配,也可以是广泛匹配。
    const struct input_device_id *id_table;

    //关联 input_handle 的信息,这样就可以与 input_dev 关联起来
    struct list_head        h_list;
    struct list_head        node;
};
```

从上面的 input handler 分析可见，其中主要内容是操作接口，并且也包含了与应用层的接口 file_operations 文件操作。

接下来是管理 input_dev 和 input_handler 关联的实体 input_handle，具体的信息如下：

```
struct input_handle {
//通常为 handler 的 private 用于进行 handler 的 event 调用时转换为实际管理实体
    void  * private;

    int open;
    //关联的设备名
    const char  * name;

    //关联 input_dev 和 input_handler
    struct input_dev  * dev;
    struct input_handler  * handler;

    //在 input_dev 的链表中
    struct list_head          d_node;
    //在 handler 的链表中
    struct list_head          h_node;
};
```

这三部分管理实体最终会根据实际情况连接成如图 6-3 所示的形式。

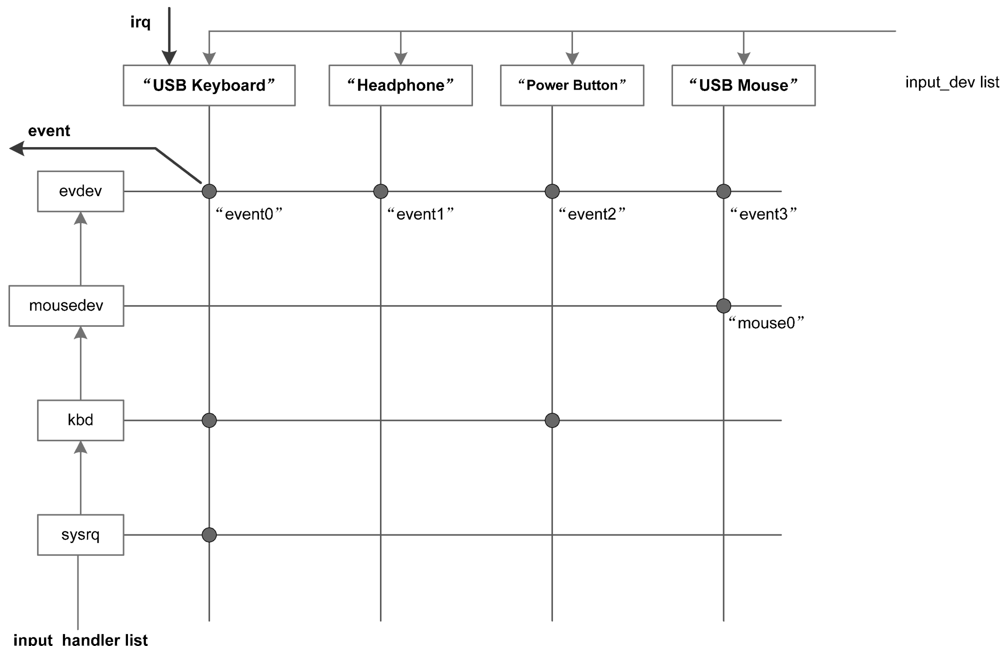


图 6-3　管理实体关系

从图 6-3 可见，中间的圆点代表 input_handle，相应的名字是设备名，而对于 input_dev 和 input_handler 都可以是一对多的关系，需要注意的是所有的设备都会绑定 evdev，也就是说 evdev 是接受所有设备绑定的。框架会有两个链表分别管理 input_dev 和 input_handle。另外如 kdb 和 sysrq 这两个 event handler 都属于内核内部的 event handler，用于内部模块对输入事件的监听。

事件是输入设备的数据流核心，整个输入设备系统的处理都是围绕这些事件进行的。不同类型的设备相应的事件也是不同的。应用层则要根据输入设备的事件进行相应的操作，为了给应用层统一的接口，输入设备框架对于输入事件进行了规范。

首先系统规范了事件类型。具体内容如下：

- EV_SYN：用于标记分段的事件，如坐标分为 x、y、z，实际是三个子事件作为一个整体，通过 SYN 可将它们组合进行处理。另在 multitouch 的输入设备根据 EV_SYN 定义了规范，将多点同时的操作分组上报。
- EV_KEY：用于支持按键的事件，主要表示按键的状态转换。
- EV_REL：相对位移事件，主要用于鼠标这类设备。
- EV_ABS：绝对位置事件，主要用于触摸屏这类设备。
- EV_MSC：杂项事件，基本是将硬件信息直接上报由应用层处理，如传送扫描码。
- EV_SW：表示一些 laptop 中二值的状态。
- EV_LED：表示 LED 的开关。
- EV_SND：表示一些单音的输出。
- EV_REP：用于自动重复的事件。
- EV_FF：强制 feedback 事件给输入设备。
- EV_PWR：开关键的相关事件。
- EV_FF_STATUS：从设备返回的强制 feedbak 的状态信息。

从这些事件可以看出，对于输入设备框架中考虑到 PC、laptop 等各种类型的输入设备，还可以反馈给设备需要的指示以及声音的相关操作，以反映用户的操作状态。典型的嵌入式设备前五种事件基本就能够涵盖相应的需求。每类事件的具体 code 信息可见 input. h，其中有详细的定义。

3. input device 和 event handler 的管理

输入设备子系统最重要的实体就是代表 input device 的 input_dev 和代表 event handler 的 input_handler。接下来看看系统是如何对它们进行管理的。

通常框架层都需要实体的注册接口，来提供添加新的实体，先来看看 input_dev 的注册细节：

```
int input_register_device(struct input_dev *dev)
{
    static atomic_t input_no = ATOMIC_INIT(0);
    struct input_handler *handler;
    const char *path;
    int error;
```

```
        /* Every input device generates EV_SYN/SYN_REPORT events. */
        //设置 SYN 事件的能力,所有的输入设备都要提供该能力
        __set_bit(EV_SYN, dev->evbit);

        /* KEY_RESERVED is not supposed to be transmitted to userspace. */
        // KEY_RESERVED 不能提供给应用层所以清除
        __clear_bit(KEY_RESERVED, dev->keybit);

        /* Make sure that bitmasks not mentioned in dev->evbit are clean. */
        input_cleanse_bitmasks(dev);

        /*
         * If delay and period are pre-set by the driver, then autorepeating
         * is handled by the driver itself and we don't do it in input.c.
         */
        //核心层提供 auto repeat 的功能,这里初始化相关的属性
        init_timer(&dev->timer);
        if(!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
                dev->timer.data = (long) dev;
                dev->timer.function = input_repeat_key;
                dev->rep[REP_DELAY] = 250;
                dev->rep[REP_PERIOD] = 33;
        }

        //框架层提供默认操作的接口
        if(!dev->getkeycode && !dev->getkeycode_new)
                dev->getkeycode_new = input_default_getkeycode;

        if(!dev->setkeycode && !dev->setkeycode_new)
                dev->setkeycode_new = input_default_setkeycode;

        dev_set_name(&dev->dev, "input%ld",
                     (unsigned long) atomic_inc_return(&input_no) - 1);

        //添加到设备模型,这里是底层的设备,还没有设备号
        error = device_add(&dev->dev);
        if(error)
                return error;
…
error = mutex_lock_interruptible(&input_mutex);
        if(error) {
                device_del(&dev->dev);
```

```
            return error;
        }

        //添加到列表中统一管理
        list_add_tail(&dev->node, &input_dev_list);

        //进行 event handler 的绑定,handler 通过框架的链表管理,之前已经注册
        list_for_each_entry(handler, &input_handler_list, node)
            input_attach_handler(dev, handler);

        input_wakeup_procfs_readers();

        mutex_unlock(&input_mutex);

        return 0;
    }
```

从代码中可见，input device 的内部属性都是为了各种类型的输入设备准备的，另外内核还提供了统一的操作接口来进行 keycode 的设置，这些基本属于标准操作。而相应设备的分配通过 input_allocate_device 来执行。

其中比较重要的就是与 event handler 的绑定，下面来分析一下具体的操作：

```
    static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
    {
        const struct input_device_id *id;
        int error;
        //检查是否匹配,返回支持的 id 号
        id = input_match_device(handler, dev);
        if(!id)
            return -ENODEV;

        //建立 input device 与具体 handler 的连接
        error = handler->connect(handler, dev, id);
        if(error && error != -ENODEV)
            printk(KERN_ERR
                "input: failed to attach handler %s to device %s, "
                "error: %d\n",
                handler->name, kobject_name(&dev->dev.kobj), error);

        return error;
    }
```

从中可见主要就进行了两个操作：检查匹配和建立连接。下面分析一下匹配的流程：

```
static const struct input_device_id *input_match_device(struct input_handler *handler,
                                                        struct input_dev *dev)
{
    const struct input_device_id *id;
    int i;

    //每个 handler 有自己的匹配表可以进行宽泛的匹配和严格的匹配
    for(id = handler->id_table; id->flags || id->driver_info; id++) {
        //通过对 id->flags 来进行严格的匹配
        //bustype 说明输入设备的连接类型,比如与 host 直接相连,还是通过蓝牙,usb 连接
        if(id->flags & INPUT_DEVICE_ID_MATCH_BUS)
            if(id->bustype != dev->id.bustype)
                continue;
        //以下要匹配厂商信息
        if(id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
            if(id->vendor != dev->id.vendor)
                continue;

        if(id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
            if(id->product != dev->id.product)
                continue;

        if(id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
        if(id->version != dev->id.version)
            continue;

        //能力匹配,如果 handler 设置了检查的能力位,则设备必须有该能力
        //event handler 的 id 表中不设置表示不检查
        MATCH_BIT(evbit,  EV_MAX);
        MATCH_BIT(keybit, KEY_MAX);
        MATCH_BIT(relbit, REL_MAX);
        MATCH_BIT(absbit, ABS_MAX);
        MATCH_BIT(mscbit, MSC_MAX);
        MATCH_BIT(ledbit, LED_MAX);
        MATCH_BIT(sndbit, SND_MAX);
        MATCH_BIT(ffbit,  FF_MAX);
        MATCH_BIT(swbit,  SW_MAX);

        //handler 具体的匹配接口,通常为空
        if(!handler->match || handler->match(handler, dev))
            return id;
```

```
    }

    return NULL;
}
```

可见在设备与 handler 的匹配中尽力进行匹配，而且允许进行宽泛的匹配，对于输入设备系统中最宽泛的匹配就是 evdev handler，相应的 id 表如下：

```
static const struct input_device_id evdev_ids[ ] = {
    { .driver_info = 1 },/* Matches all devices */
    { },            /* Terminating zero entry */
};
```

可见相应设置很简单，只要 driver_info 设置为 1 即可。

匹配之后的操作就是将 input device 与 event handle 建立连接，会调用 event handler 的 connect 接口函数，主要的工作是建立 input_handle，并通过 input_register_handle 将相关网状连接建立起来。

系统的事件是如何传送的呢？通过 input_event 来进行事件的传送，具体如下：

```
void input_event(struct input_dev *dev,
                 unsigned int type, unsigned int code, int value)
{
    unsigned long flags;

    //检查事件类型是否支持
    if(is_event_supported(type, dev->evbit, EV_MAX)) {

        spin_lock_irqsave(&dev->event_lock, flags);
        //输入是随机事件,所以可以作为系统的随机源
        add_input_randomness(type, code, value);
        //具体的将事件转发的接口
        input_handle_event(dev, NULL, type, code, value);
        spin_unlock_irqrestore(&dev->event_lock, flags);
    }
}
```

input_handle_event 会根据具体的事件类型来判断是向上层 event handler 发送还是发送给 input device 或者两者都发送，进而进行对应的发送处理（主要是调用对应的接口函数）。向 event handler 转发最终通过 input_pass_event 来实现。

```
static void input_pass_event(struct input_dev *dev,
                             struct input_handler *src_handler,
```

```
                        unsigned int type, unsigned int code, int value)
{
    struct input_handler  * handler;
    struct input_handle  * handle;

    rcu_read_lock( );

    //系统可以设置相应的输入设备单一连接 event handler,这里获得相关属性
    handle = rcu_dereference( dev -> grab) ;
    //如果单一连接直接传送事件到对应的 event handler
    if( handle)
        handle -> handler -> event( handle, type, code, value) ;
    else {
        bool filtered = false;

        //这里表示多连接的情况,遍历所有的连接点
        list_for_each_entry_rcu( handle, &dev -> h_list, d_node) {
            if( ! handle -> open)
                continue;

        //获得相应的 event handler
        handler = handle -> handler;

        //这里表示之前已经拒绝了该事件则直接跳过
        if( handler == src_handler)
            continue;

        //需要的话进行事件的过滤
        if( ! handler -> filter) {
            if( filtered)
                break;
            //将事件转发给 event handler
            handler -> event( handle, type, code, value) ;

        } else if( handler -> filter( handle, type, code, value) )
            filtered = true;
        }
    }

    rcu_read_unlock( ) ;
}
```

由于框架层提供诸如 auto repeat 等功能，所以也会使用 input_pass_event 来进行事件的传送。

为了设备操作方便，系统还基于 input_event 进行了封装，为不同类型的事件提供了事件转发操作的标准接口，接口如下：

```
static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)
static inline void input_report_rel(struct input_dev *dev, unsigned int code, int value)
static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)
static inline void input_report_ff_status(struct input_dev *dev, unsigned int code, int value)
static inline void input_report_switch(struct input_dev *dev, unsigned int code, int value)
static inline void input_sync(struct input_dev *dev)
static inline void input_mt_sync(struct input_dev *dev)
static inline void input_mt_slot(struct input_dev *dev, int slot)
```

可见框架为了开发者的方便做了不少工作，这样统一的接口可以降低代码冗余度。整体上框架对 input device 和 event handler 的管理除了实体的管理外还负责事件的转发工作。

4. 设备号管理及设备文件创建

对为应用层提供服务的框架，设备文件相关的管理是十分重要的，这里分析输入设备框架相关的实现。首先作为字符设备的子类型输入设备会统一注册文件操作接口，用于输入设备打开的管理。具体就是 input_fops：

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
    .llseek = noop_llseek,
};
```

从 input_fops 中可见其主要的操作就是 open，具体的分析如下：

```
static int input_open_file(struct inode *inode, struct file *file)
{
    struct input_handler *handler;
    const struct file_operations *old_fops, *new_fops = NULL;
    int err;

    err = mutex_lock_interruptible(&input_mutex);
    if(err)
        return err;

    /* No load-on-demand here? */
    //通过全局的 input_table 来管理框架内部的设备,实际指向的是 event handler
    handler = input_table[iminor(inode) >> 5];
```

```
        if(handler)
            new_fops = fops_get(handler->fops);

        mutex_unlock(&input_mutex);

        /*
         * That's _really_ odd. Usually NULL ->open means "nothing special",
         * not "no device". Oh, well...
         */
        if(!new_fops || !new_fops->open) {
            fops_put(new_fops);
            err = -ENODEV;
            goto out;
        }

        //重载文件的操作接口
        old_fops = file->f_op;
        file->f_op = new_fops;

        //用新的接口打开,如果失败则恢复文件操作接口
        err = new_fops->open(inode, file);
        if(err) {
            fops_put(file->f_op);
            file->f_op = fops_get(old_fops);
        }
        fops_put(old_fops);
    out:
        return err;
    }
```

从分析可见，输入设备系统对应的设备文件使用统一的主设备号，而子设备号是与 event handler 相关的，不同的 handler 管理不同的子设备号。再来看看 event handler 的注册接口：

```
    int input_register_handler(struct input_handler *handler)
    {
        struct input_dev *dev;
        int retval;

        retval = mutex_lock_interruptible(&input_mutex);
        if(retval)
            return retval;
```

```
        INIT_LIST_HEAD(&handler->h_list);

        //包含文件操作接口,说明该 handler 会对用户开放
        if(handler->fops != NULL) {
            //加入 input_table,用来在打开文件时做重定向操作
            if(input_table[handler->minor >> 5]) {
                retval = -EBUSY;
                goto out;
            }
            //这里可见一个 handler 管理 32 个设备
            input_table[handler->minor >> 5] = handler;
        }

        list_add_tail(&handler->node, &input_handler_list);

        list_for_each_entry(dev, &input_dev_list, node)
            input_attach_handler(dev, handler);

        input_wakeup_procfs_readers();

        out:
            mutex_unlock(&input_mutex);
            return retval;
    }
```

这里不仅包括对 event handler 的管理，由于 event handler 还要作为用户的接口，还要进行子设备的分配，只是输入框架已经提前预分配好了，每个 handler 管理 32 个设备。

最后再来看具体的 evdev handler 的 connect 操作：

```
    static int evdev_connect(struct input_handler *handler, struct input_dev *dev,
                             const struct input_device_id *id)
    {
        struct evdev *evdev;
        int minor;
        int error;

        //内部管理的子设备号分配情况
        for(minor = 0; minor < EVDEV_MINORS; minor++)
            if(! evdev_table[minor])
                break;

        if(minor == EVDEV_MINORS) {
```

```
        printk(KERN_ERR "evdev: no more free evdev devices\n");
        return -ENFILE;
    }

    //得到子设备号后,进行管理实体的分配
    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
    if(! evdev)
        return -ENOMEM;

    //属性的初始化
    INIT_LIST_HEAD(&evdev->client_list);
    spin_lock_init(&evdev->client_lock);
    mutex_init(&evdev->mutex);
    //等待队列管理操作的进程
    init_waitqueue_head(&evdev->wait);

    dev_set_name(&evdev->dev, "event%d", minor);
    evdev->exist = true;
    //记录内部的子设备号
    evdev->minor = minor;

    //这里是与底层 input device 连接点的属性赋值,便于建立系统内部的关联
    evdev->handle.dev = input_get_device(dev);
    evdev->handle.name = dev_name(&evdev->dev);
    evdev->handle.handler = handler;
    evdev->handle.private = evdev;

    //真正的设备号及设备的初始化,这里是对应于用户的设备
    evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);
    //功能类型设备
    evdev->dev.class = &input_class;
    //层次上要对应于 input device 的设备
    evdev->dev.parent = &dev->dev;
    evdev->dev.release = evdev_free;
    device_initialize(&evdev->dev);

    //建立系统内部的关联
    error = input_register_handle(&evdev->handle);
    if(error)
        goto err_free_evdev;

    //内部管理设备
```

```
        error = evdev_install_chrdev( evdev) ;
        if( error)
            goto err_unregister_handle;

        //这里通知系统加入新的设备,相应的会有应用层建立设备文件
        error = device_add( &evdev -> dev) ;
        if( error)
            goto err_cleanup_evdev;

        return 0;

    err_cleanup_evdev:
        evdev_cleanup( evdev) ;
    err_unregister_handle:
        input_unregister_handle( &evdev -> handle) ;
    err_free_evdev:
        put_device( &evdev -> dev) ;
        return error;
    }
```

可见具体的设备号管理还是在 event handler 内部进行的。从实现中设备的层次设置可以了解到，设备的层次关系的添加是逐层进行的，这部分工作由各层独立完成。

6.1.3　输入设备应用层操作及框架适配

应用层对输入设备的基本使用主要是集中在读取相应的事件方面，具体事件传给哪个应用则是属于应用框架的范畴，与具体的 UI 控制紧密关联。

先来看看简单的测试程序，对设备的输入进行检查并在控制台上进行显示。

```
    int main(int argc, char ** argv)
    {
        int fd, rd, i, j, k;
        struct input_event ev[64];
        int version;
        unsigned short id[4];
        unsigned long bit[EV_MAX][NBITS(KEY_MAX)];
        char name[256] = "Unknown";
        int abs[5];
        int counter =0;
        int iterations =0;

        if(argc < 3) {
            printf("Usage: evtest /dev/input/eventX <iterations> \n");
```

```
        printf("Where X = input device number\n");
        return 1;
    }

    //打开文件
    if((fd = open(argv[argc - 2], O_RDONLY)) < 0) {
        perror("evtest: cannot open node");
        return 1;
    }

    iterations = atoi(argv[argc - 1]);
    printf("Number of iterations %d\n", iterations);
    if(iterations < 1) {
        perror("evtest: number of iterations shall be >1\n");
        return 1;
    }

    //获得基本信息
    if(ioctl(fd, EVIOCGVERSION, &version)) {
        perror("evtest: can't get version");
        return 1;
    }

    printf("Input driver version is %d.%d.%d\n",
        version >> 16, (version >> 8) & 0xff, version & 0xff);

    ioctl(fd, EVIOCGID, id);
    printf("Input device ID: bus 0x%x vendor 0x%x product 0x%x version 0x%x\n",
        id[ID_BUS], id[ID_VENDOR], id[ID_PRODUCT], id[ID_VERSION]);

    ioctl(fd, EVIOCGNAME(sizeof(name)), name);
    printf("Input device name: \"%s\"\n", name);

    memset(bit, 0, sizeof(bit));
    ioctl(fd, EVIOCGBIT(0, EV_MAX), bit[0]);
    printf("Supported events:\n");

    //根据支持的事件类型获得设备能力并进行相应的显示
    for(i = 0; i < EV_MAX; i++)
        if(test_bit(i, bit[0])) {
            printf("  Event type %d(%s)\n", i, events[i] ? events[i] : "?");
            if(!i) continue;
```

```
            ioctl(fd, EVIOCGBIT(i, KEY_MAX), bit[i]);
            for(j = 0; j < KEY_MAX; j++)
                if(test_bit(j, bit[i])) {
                    printf("Event code %d(%s)\n", j, names[i] ? (names[i][j] ? names
[i][j] : "?") : "?");
                    if(i == EV_ABS) {
                        ioctl(fd, EVIOCGABS(j), abs);
                        for(k = 0; k < 5; k++)
                            if((k < 3) || abs[k])
                                printf("        %s %6d\n", absval[k], abs[k]);
                    }
                }
            }

        printf("Interacting %d times... (interrupt to exit)\n",
                iterations);

    //迭代读取输入事件,输出事件信息,这里是 block 同步读取
    while(counter < iterations) {
        printf("Info : %d of %d iterations\n", counter,
                iterations);
        rd = read(fd, ev, sizeof(struct input_event) * 64);

        if(rd < (int) sizeof(struct input_event)) {
            printf("yyy\n");
            perror("\nevtest: error reading");
            return 1;
        }

        for(i = 0; i < rd / sizeof(struct input_event); i++)

            if(ev[i].type == EV_SYN) {
                printf("Event: time %ld.%06ld, ------------ %s ---------\n",
                    ev[i].time.tv_sec, ev[i].time.tv_usec, ev[i].code ? "Config Sync" : "
                    Report Sync" );
            } else if(ev[i].type == EV_MSC && ev[i].code == MSC_RAW) {
                printf("Event: time %ld.%06ld, type %d(%s), code %d(%s), value %02x\n",
                    ev[i].time.tv_sec, ev[i].time.tv_usec, ev[i].type,
                    events[ev[i].type] ? events[ev[i].type] : "?",
                    ev[i].code,
                    names[ev[i].type] ? (names[ev[i].type][ev[i].code] ? names[ev[i]
                    .type][ev[i].code] : "?") : "?",
```

```
                        ev[i].value);
                } else {
                    printf("Event: time %ld.%06ld, type %d(%s), code %d(%s), value %d\n",
                        ev[i].time.tv_sec, ev[i].time.tv_usec, ev[i].type,
                        events[ev[i].type] ? events[ev[i].type] : "?",
                        ev[i].code,
                        names[ev[i].type] ? (names[ev[i].type][ev[i].code] ? names[ev[i]
                        .type][ev[i].code] : "?") : "?",
                        ev[i].value);
                }
            counter++;
        }
        return 0;
    }
```

从简单的测试代码可见，输入设备的操作逻辑基本上是打开设备，检查设备能力，应用需要根据能力进行相关的配置。最后就是读取输入事件转发给系统的框架。

接下来看看 Android 是如何进行相关适配的。对于 Android 框架来说，输入设备的相关操作是在 EventHub 中实现的，首先需要检查所有的输入文件，其是通过 inotify 来实现的。

在 EventHub 的构造函数中会对整个/dev/input 进行监视。

```
mINotifyFd = inotify_init();
int result = inotify_add_watch(mINotifyFd, DEVICE_PATH, IN_DELETE | IN_CREATE);
```

从中可见，对设备创建和删除都进行监控，这样系统会对输入设备有全局的把控，每当有相应的事件发生时都会调用 EventHub::readNotifyLocked，其中会有如下代码

```
while(res >= (int)sizeof(*event)) {
    event = (struct inotify_event *)(event_buf + event_pos);
    if(event->len) {
        strcpy(filename, event->name);
        if(event->mask & IN_CREATE) {
            openDeviceLocked(devname);
        } else {
            ALOGI("Removing device '%s' due to inotify event\n", devname);
            closeDeviceByPathLocked(devname);
        }
    }
    event_size = sizeof(*event) + event->len;
    res -= event_size;
    event_pos += event_size;
}
```

一旦发现是 create 文件的时候会进行打开设备的操作，其中 openDeviceLocked 会根据类似之前测试代码的 ioctl 方式获得输入设备的信息，并进行相关的属性配置，然后通过如下代码以 poll 的方式来监测设备的输入：

```
struct epoll_event eventItem;
memset(&eventItem, 0, sizeof(eventItem));
eventItem.events = EPOLLIN;
eventItem.data.u32 = deviceId;
if(epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, &eventItem)) {
    ALOGE("Could not add device fd to epoll instance.  errno = %d", errno);
    delete device;
    return -1;
}
```

设备的输入则是在 EventHub：：getEvents 中等待事件的上报，代码如下

```
int pollResult = epoll_wait(mEpollFd, mPendingEventItems, EPOLL_MAX_EVENTS, timeoutMillis);
```

事件上报后，getEvents 会读取并检查确认具体的有效事件数目后返回，后续操作就是上报给应用框架层进行处理。这样就实现了输入设备的整体适配。

6.1.4 TI 芯片输入设备相关实现详解

对 DM 3730 的开发板，相关的输入设备是电源管理芯片中的键盘矩阵，这里对相关的输入设备驱动进行分析。

在硬件介绍中已经见到 DM 3730 与电源管理芯片是通过 I^2C 总线进行连接的，所以该键盘驱动需要建立在 I^2C 总线操作之上。

相应的驱动首先是 platform_driver，细节如下：

```
static struct platform_driver twl4030_kp_driver = {
    .probe = twl4030_kp_probe,
    .remove = __devexit_p(twl4030_kp_remove),
    .driver = {
        .name = "twl4030_keypad",
        .owner = THIS_MODULE,
    },
};
```

其中 probe 接口是了解驱动的入口，它是在总线进行匹配之后进行的，主要进行根据设备信息申请资源以及驱动相关的初始化。下面进行详细分析：

```
static int __devinit twl4030_kp_probe(struct platform_device *pdev)
{
    struct twl4030_keypad_data *pdata = pdev->dev.platform_data;
```

```
const struct matrix_keymap_data *keymap_data = pdata->keymap_data;
struct twl4030_keypad *kp;
struct input_dev *input;
u8 reg;
int error;

//传入的设备信息中键盘矩阵的维度信息不能为0
if(!pdata || !pdata->rows || !pdata->cols ||
    pdata->rows > TWL4030_MAX_ROWS || pdata->cols > TWL4030_MAX_COLS)
{
    dev_err(&pdev->dev, "Invalid platform_data\n");
    return -EINVAL;
}

//分配驱动中设备的管理实体
kp = kzalloc(sizeof(*kp), GFP_KERNEL);
//分配输入框架的 input device 实体
input = input_allocate_device();
if(!kp || !input) {
    error = -ENOMEM;
    goto err1;
}

/* Get the debug Device */
kp->dbg_dev = &pdev->dev;
//与输入设备框架建立关联
kp->input = input;

//记录设备信息
kp->n_rows = pdata->rows;
kp->n_cols = pdata->cols;
//获得中断号,由于电源管理芯片是通过引脚对 DM 3730 进行中断,这里为
//其提供单独的中断号
kp->irq = platform_get_irq(pdev, 0);

/* setup input device */
//设置键盘事件类型
__set_bit(EV_KEY, input->evbit);

/* Enable auto repeat feature of Linux input subsystem */
//根据设备配置是否自动重复
if(pdata->rep)
```

```
        __set_bit(EV_REP, input->evbit);

    //设置 MSC 类型,提供扫描码
    input_set_capability(input, EV_MSC, MSC_SCAN);

    //根据设备信息设置 input device 的相关属性
    input->name = "TWL4030 Keypad";
    input->phys = "twl4030_keypad/input0";
    input->dev.parent = &pdev->dev;

    input->id.bustype = BUS_HOST;
    input->id.vendor = 0x0001;
    input->id.product = 0x0001;
    input->id.version = 0x0003;

    input->keycode = kp->keymap;
    input->keycodesize = sizeof(kp->keymap[0]);
    input->keycodemax = ARRAY_SIZE(kp->keymap);

    matrix_keypad_build_keymap(keymap_data, TWL4030_ROW_SHIFT,
                input->keycode, input->keybit);

    //注册 input device
    error = input_register_device(input);
    if(error) {
        dev_err(kp->dbg_dev,
            "Unable to register twl4030 keypad device\n");
        goto err1;
    }

    error = twl4030_kp_program(kp);
    if(error)
        goto err2;

    //由于通过 I2C 总线进行相关操作,所以这里设置为线程执行中断
    error = request_threaded_irq(kp->irq, NULL, do_kp_irq,
            0, pdev->name, kp);
    if(error) {
        dev_info(kp->dbg_dev, "request_irq failed for irq no = %d\n",
            kp->irq);
    goto err2;
```

```
    }

        //这里使能键盘上报中断
        reg = (u8) ~(KEYP_IMR1_KP | KEYP_IMR1_TO);
        if(twl4030_kpwrite_u8(kp, reg, KEYP_IMR1)) {
            error = -EIO;
            goto err3;
        }

        //将私有信息加入 platform device 中,以便后续操作实例化的需要
        platform_set_drvdata(pdev, kp);
        return 0;
    ...
    }
```

从中可见，主要的流程就是针对输入设备框架进行 input_dev 的相关初始化和注册，另外就是注册中断。相应的 platform_device 是通过 add_numbered_child 来创建的，原因是由于电源管理芯片实际包含音频处理、键盘、USB phy 等多种功能，在 Linux 内核中将这类设备归为 mfd（即 multi function device），实际还是一个一个单独的设备，但是通过统一的接口进行操作，具体就不进行详述了。

回到驱动，当有按键输入时就会触发中断，进而执行驱动的中断处理函数，内容如下：

```
    static irqreturn_t do_kp_irq(int irq, void *_kp)
    {
        struct twl4030_keypad *kp = _kp;
        u8 reg;
        int ret;

        /* Read & Clear TWL4030 pending interrupt */
        ret = twl4030_kpread(kp, &reg, KEYP_ISR1, 1);

        if(ret >= 0 &&(reg & KEYP_IMR1_KP))
            twl4030_kp_scan(kp, false);
        else
            twl4030_kp_scan(kp, true);

        return IRQ_HANDLED;
    }
```

其中主要的操作就是读取扫描码，具体由 twl4030_kp_scan 来实现。

```
static void twl4030_kp_scan(struct twl4030_keypad *kp, bool release_all)
{
    struct input_dev *input = kp->input;
    u16 new_state[TWL4030_MAX_ROWS];
    int col, row;

    if(release_all)
        memset(new_state, 0, sizeof(new_state));
    else {
        /* check for any changes */
        //检测键盘矩阵的状态变化,通过 I2C 总线进行
        int ret = twl4030_read_kp_matrix_state(kp, new_state);

        if(ret < 0)/* panic ... */
            return;

        if(twl4030_is_in_ghost_state(kp, new_state))
            return;
    }

    /* check for changes and print those */
    //根据变化获得扫描码
    for(row = 0; row < kp->n_rows; row++) {
        int changed = new_state[row] ^ kp->kp_state[row];

        if(!changed)
            continue;

        /* Extra column handles "all gnd" rows */
        for(col = 0; col < kp->n_cols + 1; col++) {
            int code;

            if(!(changed &(1 << col)))
                continue;

            dev_dbg(kp->dbg_dev, "key [%d:%d] %s\n", row, col,
                (new_state[row] &(1 << col)) ?
                "press" : "release");
            //获得扫描码
            code = MATRIX_SCAN_CODE(row, col, TWL4030_ROW_SHIFT);
            //上报扫描码
            input_event(input, EV_MSC, MSC_SCAN, code);
```

```
                //上报预设的键值
                input_report_key(input, kp->keymap[code],
                new_state[row] &(1 << col));
            }
            //记录新的状态
            kp->kp_state[row] = new_state[row];
        }
        //一次完整的事件所以上报 SYN 事件
        input_sync(input);
    }
```

对键盘来说按下与抬起都是事件，所以在进行事件检查时是根据变化进行确认，而不是简单的根据值进行确认。具体获得相关信息的方法都是通过类似 twl4030_read_kp_matrix_state 的操作完成的，其中会进行实际的 I^2C 总线操作，具体的细节会在总线部分进行介绍。在驱动获得硬件的信息后就是向输入设备框架汇报相应的事件，框架会上传给 event handler，最终被用户读取。这样输入的通道就从硬件到应用层打通了。

6.1.5 输入设备电源管理相关说明

关于具体的电源管理，先来看看框架层提供了哪些内容。由于这是功能型设备，所以电源管理主要是在设备层面，来看看 input device 分配时的具体操作：

```
dev->dev.type = &input_dev_type;
dev->dev.class = &input_class;
```

主要是设置了设备类型以及设备功能类，下面来看看设备类型 input_dev_type 的内容：

```
static struct device_type input_dev_type = {
    .groups = input_dev_attr_groups,
    .release = input_dev_release,
    .uevent = input_dev_uevent,
#ifdef CONFIG_PM
    .pm = &input_dev_pm_ops,
#endif
};
```

可见其中包含电源管理的功能 input_dev_pm_ops，细节如下：

```
static const struct dev_pm_ops input_dev_pm_ops = {
    .suspend = input_dev_suspend,
    .resume = input_dev_resume,
    .poweroff = input_dev_suspend,
    .restore = input_dev_resume,
};
```

看一下具体的接口内容：

```
static int input_dev_suspend(struct device *dev)
{
    struct input_dev *input_dev = to_input_dev(dev);

    mutex_lock(&input_dev->mutex);

    //如果有使用者进行输入设备警示类的操作,需要设备支持
    if(input_dev->users)
        input_dev_toggle(input_dev, false);

    mutex_unlock(&input_dev->mutex);

    return 0;
}

static int input_dev_resume(struct device *dev)
{
    struct input_dev *input_dev = to_input_dev(dev);

    //显示的 reset 设备,以便后续的操作
    input_reset_device(input_dev);

    return 0;
}
```

可见在框架层，并没有进行实际的电源管理操作，而是起到提示以及保证设备正常使用的作用。

对于物理设备的电源管理部分，以 GPIO 键盘为例，其中提供了电源管理相关的接口如下：

```
static const struct dev_pm_ops gpio_keys_pm_ops = {
    .suspend = gpio_keys_suspend,
    .resume = gpio_keys_resume,
};
```

这里是系统待机（suspend）操作和恢复（resume）操作的接口，下面来看看具体的细节：

```
static int gpio_keys_suspend(struct device *dev)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
    int i;
```

```
        //检查是否设置了设备可以唤醒
        if(device_may_wakeup(&pdev->dev)) {
            for(i=0; i < pdata->nbuttons; i++) {
                struct gpio_keys_button *button = &pdata->buttons[i];
                if(button->wakeup) {
                    //该 GPIO 为唤醒键则设置相应的中断唤醒能力
                    int irq = gpio_to_irq(button->gpio);
                    enable_irq_wake(irq);
                }
            }
        }

        return 0;
    }

    static int gpio_keys_resume(struct device *dev)
    {
        struct platform_device *pdev = to_platform_device(dev);
        struct gpio_keys_drvdata *ddata = platform_get_drvdata(pdev);
        struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
        int i;

        for(i=0; i < pdata->nbuttons; i++) {

            struct gpio_keys_button *button = &pdata->buttons[i];
            //关闭相应的唤醒能力
            if(button->wakeup && device_may_wakeup(&pdev->dev)) {
                int irq = gpio_to_irq(button->gpio);
                disable_irq_wake(irq);
            }

            //检查 GPIO 键的状态,需要的话进行上报
            gpio_keys_report_event(&ddata->data[i]);
        }
        //报告 SYN 表示完整的事件
        input_sync(ddata->input);

        return 0;
    }
```

具体设备根据是否唤醒的需要进行相应的设置，从而完成电源管理相关功能。

6.2　帧缓冲（frame buffer）

6.2.1　帧缓冲设备需求

对计算机以及嵌入式设备来说，显示设备都是重要的输出设备，其承载了系统针对用户的 UI 输出，对用户体验至关重要。从用户体验来看，显示设备的分辨率是重要的一个指标，分辨率越高显示效果越好，随着技术的发展，显示设备的分辨率也是越来越高。主要的分辨率的指标如图 6–4 所示。

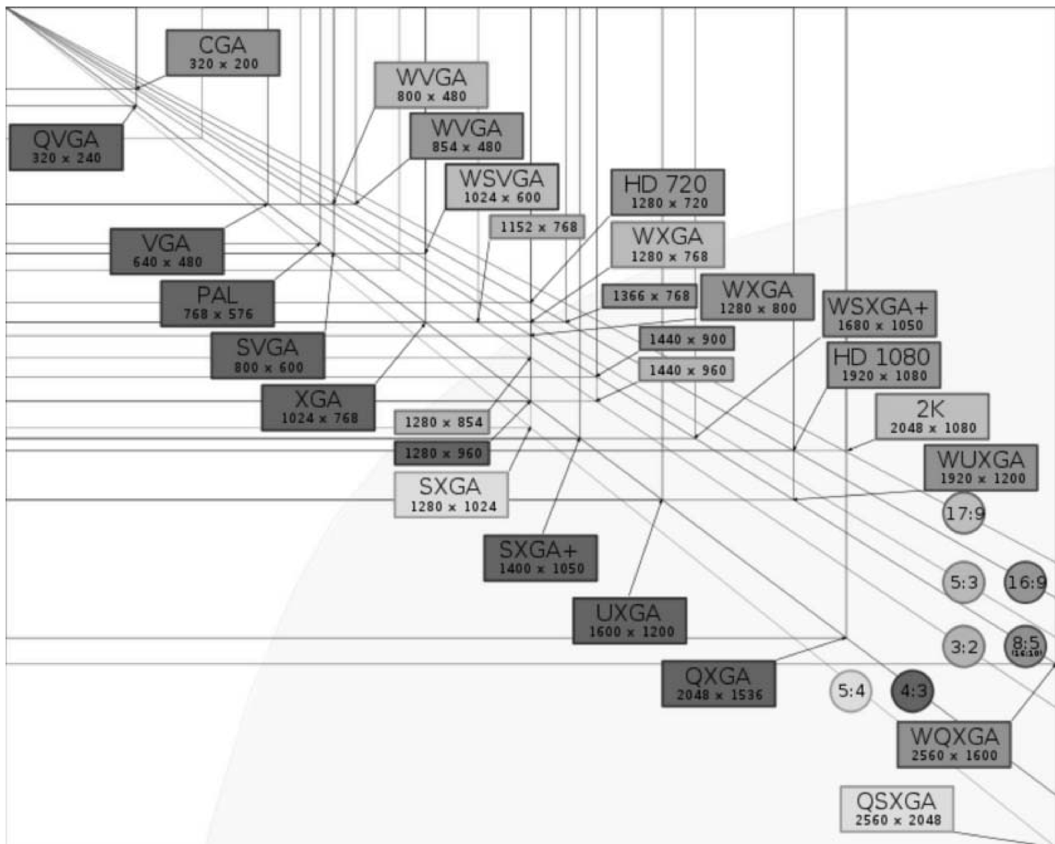


图 6–4　分辨率指标

从图 6–4 可见，目前各种显示分辨率，不仅大小不同，长宽比（圆圈中的比值表示长宽比）也是各种各样的，这些分辨率主要有两种不同的标准，一种是 4:3 的 PC 相关的显示器标准，另外一种是 16:9 的 TV 相关的标准。对显示驱动基本的需求就是能够支持这些各种不同分辨率和长宽比的显示设备，并对这些设备进行相应的控制，使得系统能有较好的输出效果。

随着技术的发展，显示器硬件也在不断地变化，作为外部显示设备，与处理器总是通过某种接口进行连接，一方面是标准的不同，另一方面是技术的发展。由于处理器到显示器的硬件连接方式的不同与变化，所以对显示驱动来说，还要能够支持这些不同的硬件的连接方式，并且将这些差别保存在内部，使得用户感受不到这些不同，给用户统一的体验。

以上主要是从基本硬件的角度产生的需求，而对用户体验的追求也不断提升显示硬件的技术水平，比如透视效果等。这也就需要在驱动层面支持相应的效果设置。如果从用户体验的角度考虑扩展就会增加需求，比如能够支持分屏显示、扩展显示、支持显示器动态切换等，这些类型的需求同样是整个系统的需求，所以在具体的实现方案和层次方面可以不必在设备层支持，究竟如何实现则要从硬件能力和整体的角度考虑，这就仁者见仁、智者见智了。

6. 2. 2　帧缓冲框架解析

Linux 内核中帧缓冲对应的就是显示输出设备，它是从一个包含了完整显示帧数据的内存缓冲区来驱动视频显示器。主要是将显示设备抽象为显示的内容，通过显示内容来对显示设备进行操作。从层次上考虑对用户可见的是不同的显示内容，而物理显示设备的控制则由驱动进行处理，当然用户不能对物理设备的参数一无所知，由于应用层控制显示内容和效果，所以相应的也需要获得显示参数。而从驱动的角度考虑，只要将显示内容统一的看作数据就可以在抽象操作上将分辨率等物理参数的不同屏蔽，毕竟那些参数偏向于控制显示设备的信息。

整体的 frame buffer 框架如图 6-5 所示。

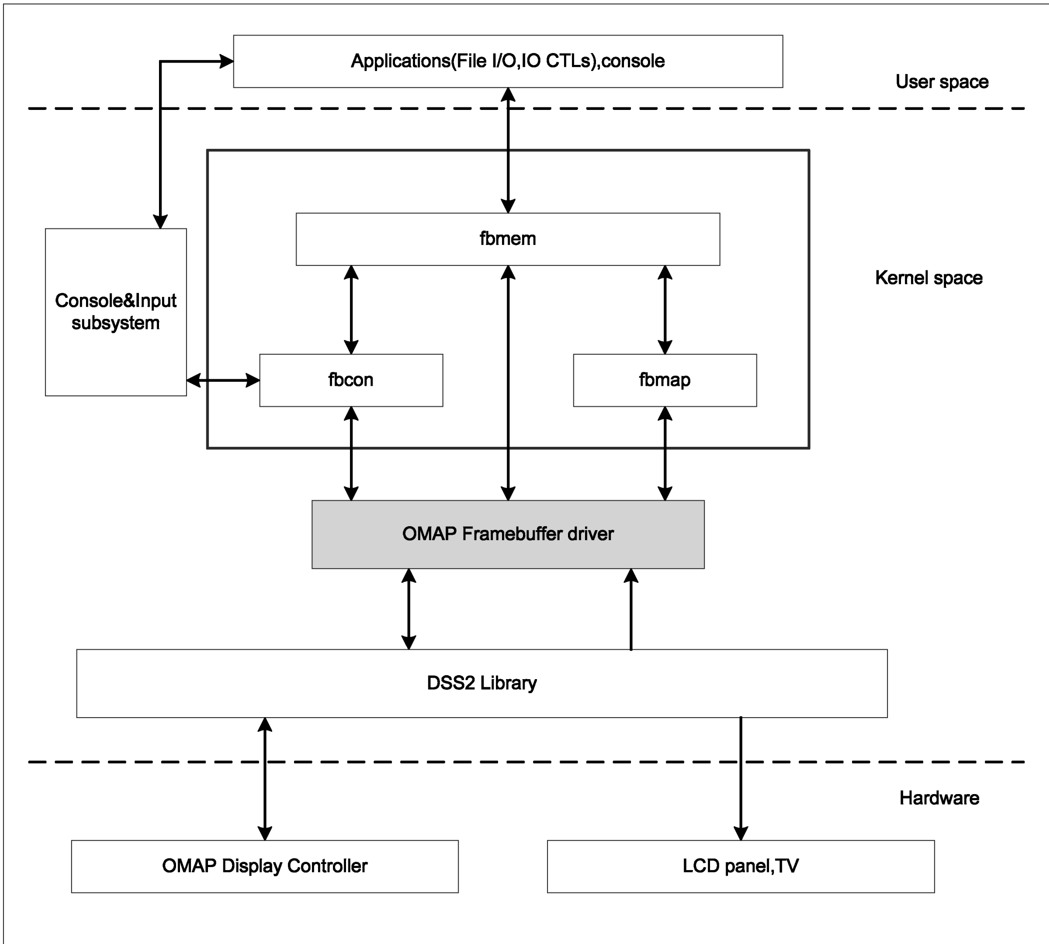


图 6-5　frame buffer 框架

从图6-5中可见，对框架来说，由于显示设备还可以显示控制台的信息，所以专门提供fbcon来进行统一的相关操作，并与console模块相结合，而fbcmap存在的原因是由于老的设备内存很少（由于当时价格昂贵），而显示设备每个像素的色彩相对丰富，所以建立一个映射表，使得用较少的信息表示颜色在内存的色彩值，并与显示设备的色彩值之间建立映射，以达到更好的显示效果。这两部分使用并不多，所以不做介绍。框架的主要部分是fbmem，而相应与驱动的关系也是很直接的。

了解frame buffer的框架还要先从为用户提供的接口开始。frame buffer提供了相应的文件操作接口fb_fops，下面来看看具体内容：

```
static const struct file_operations fb_fops = {
        .owner = THIS_MODULE,
        .read =         fb_read,
        .write          = fb_write,
        .unlocked_ioctl = fb_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl = fb_compat_ioctl,
#endif
        .mmap           = fb_mmap,
        .open           = fb_open,
        .release = fb_release,
#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
        .get_unmapped_area = get_fb_unmapped_area,
#endif
#ifdef CONFIG_FB_DEFERRED_IO
        .fsync = fb_deferred_io_fsync,
#endif
        .llseek = default_llseek,
};
```

对于直接对应驱动的框架，只要了解相应的open操作就能明确框架管理的实体是什么。下面来看看fb_open，调用fb_open说明文件的相关操作已经重载为frame buffer框架的文件操作fb_fops。

```
static int
fb_open(struct inode * inode,struct file * file)
__acquires(&info->lock)
__releases(&info->lock)
{
        int fbidx = iminor(inode);
        //这里 fb_info 就是管理的核心实体
        struct fb_info * info;
        int res = 0;
```

```
        if(fbidx >= FB_MAX)
            return -ENODEV;
        //根据子设备号获得相应的管理实体
        info = registered_fb[fbidx];
        //没有的话实体加载驱动
        if(! info)
            request_module("fb%d",fbidx);
        //再检查一遍
        info = registered_fb[fbidx];
        //这里没有就真的没有了
        if(!info)
            return -ENODEV;
        mutex_lock(&info->lock);
        if(!try_module_get(info->fbops->owner)){
            res = -ENODEV;
            goto out;
        }
        //向 vfs 加入 frame buffer 框架的私有信息,相当于实体化
        file->private_data = info;
        //进行具体驱动的 open 操作
        if(info->fbops->fb_open){
            res = info->fbops->fb_open(info,1);
            if(res)
                module_put(info->fbops->owner);
        }
    #ifdef CONFIG_FB_DEFERRED_IO
        if(info->fbdefio)
            fb_deferred_io_open(info,inode,file);
    #endif
    out:
        mutex_unlock(&info->lock);
        return res;
    }
```

从代码中可以了解，frame buffer 框架最重要的实体就是 fb_info，其中涉及驱动的操作接口 fbops。接下来分析一下 fb_info：

```
    struct fb_info{
        //在 frame buffer 框架管理的 index 号
        int node;
        int flags;
        struct mutex lock;                  /* Lock for open/release/ioctl funcs */
```

```
    struct mutex mm_lock;                   /* Lock for fb_mmap and smem_* fields */
    //显示相关的可变信息
    struct fb_var_screeninfo var;           /* Current var */
    //显示相关的固定信息
    struct fb_fix_screeninfo fix;           /* Current fix */
    struct fb_monspecs monspecs;            /* Current Monitor specs */
    struct work_struct queue;               /* Framebuffer event queue */
    struct fb_pixmap pixmap;                /* Image hardware mapper */
    //支持硬件叠加鼠标,这里是鼠标的像素信息
    struct fb_pixmap sprite;                /* Cursor hardware mapper */
    //color map 信息
    struct fb_cmap cmap;                    /* Current cmap */
    //支持模式的列表
    struct list_head modelist;              /* mode list */
    //当前的显示模式
    struct fb_videomode *mode;              /* current mode */
...
    //设备驱动的操作
    struct fb_ops *fbops;
    //设备模型的层次
    struct device *device;                  /* This is the parent */
    struct device *dev;                     /* This is this fb device */
    int class_flag;                         /* private sysfs flags */
#ifdef CONFIG_FB_TILEBLITTING
    //高级的设备支持瓦片级别的操作接口,一个瓦片是 n×n 的矩阵,可以不是连续内存
    struct fb_tile_ops *tileops;            /* Tile Blitting */
#endif
    //frame buffer 设备的显存基地址
    char __iomem *screen_base;              /* Virtual address */
    unsigned long screen_size;              /* Amount of ioremapped VRAM or 0 */
    void *pseudo_palette;                   /* Fake palette of 16 colors */
#define FBINFO_STATE_RUNNING    0
#define FBINFO_STATE_SUSPENDED    1
    u32 state;                              /* Hardware state i.e suspend */
...
};
```

其中最重要的三个属性分别是 var、fix 和 fbops，分别代表可变信息、固定信息和驱动操作接口。先来看看可变信息：

```
struct fb_var_screeninfo{
    //当前显示的分辨率大小
```

```
    __u32 xres;                    /* visible resolution */
    __u32 yres;
    //根据分配的 frame buffer 空间计算的整个分辨率
    __u32 xres_virtual;            /* virtual resolution */
    __u32 yres_virtual;
    //当前在 frame buffer 空间中根据 virtual resolution 的偏移
    __u32 xoffset;                 /* offset from virtual to visible */
    __u32 yoffset;                 /* resolution */

    //每个像素所占 bit 数目
    __u32 bits_per_pixel;          /* guess what */
    __u32 grayscale;               /* != 0 Graylevels instead of colors */

    //RGB 以及透传参数在像素 bits 中的布局信息
    struct fb_bitfield red;        /* bitfield in fb mem if true color, */
    struct fb_bitfield green;      /* else only length is significant */
    struct fb_bitfield blue;
    struct fb_bitfield transp;     /* transparency */

    __u32 nonstd;                  /* != 0 Non standard pixel format */


    __u32 activate;                /* see FB_ACTIVATE_* */

    __u32 height;                  /* height of picture in mm */
    __u32 width;                   /* width of picture in mm */

    __u32 accel_flags;             /* (OBSOLETE) see fb_info.flags */

    /* Timing: All values in pixclocks, except pixclock (of course) */
    //这里是硬件信号的信息主要包括像素时钟、前消隐、后消隐以及水平同步长度
    //垂直同步长度等信息
    __u32 pixclock;                /* pixel clock in ps (pico seconds) */
    __u32 left_margin;             /* time from sync to picture */
    __u32 right_margin;            /* time from picture to sync */
    __u32 upper_margin;            /* time from sync to picture */
    __u32 lower_margin;
    __u32 hsync_len;               /* length of horizontal sync */
    __u32 vsync_len;               /* length of vertical sync */
    __u32 sync;                    /* see FB_SYNC_* */
    __u32 vmode;                   /* see FB_VMODE_* */
    __u32 rotate;                  /* angle we rotate counter clockwise */
    __u32 reserved[5];             /* Reserved for future compatibility */
};
```

fb_var_screeninfo 会在 frame buffer 的操作中发生改变，其中主要分辨率的关系如图 6-6 所示。在实际显示过程中通过调整这些参数可以显示不同的内容，也是利用该部分实现多帧操作，提高整个的显示效果。

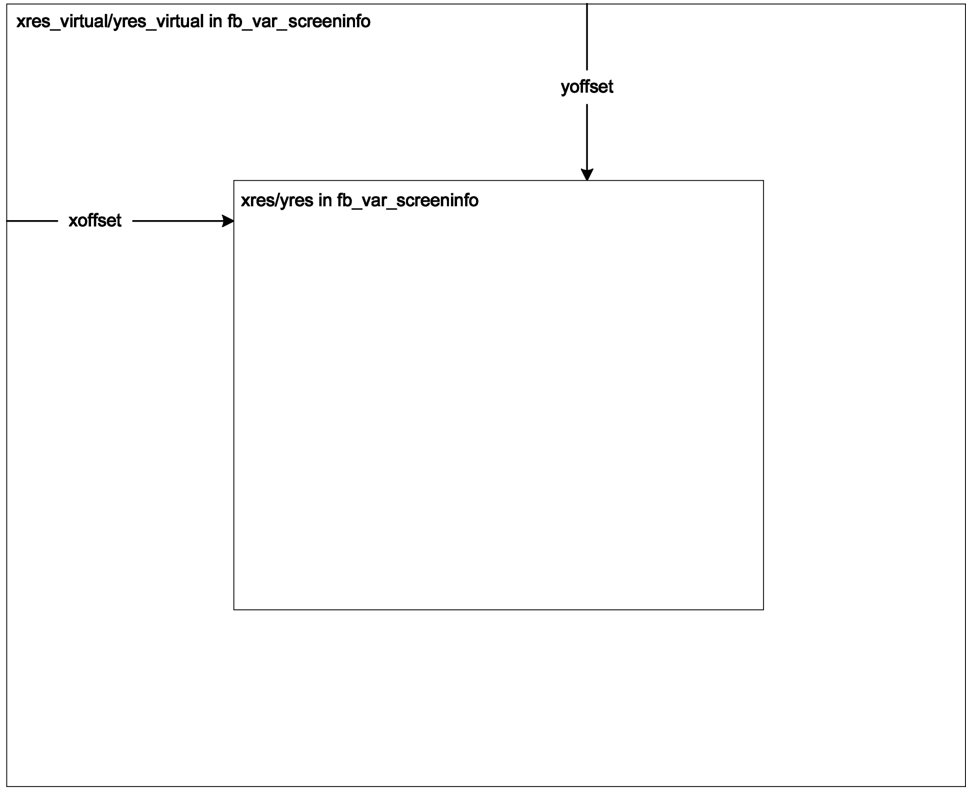


图 6-6　可变参数中分辨率的关系

硬件信息的含义如图 6-7 所示。图 6-6 引自《DM 3730 芯片手册》中第 1722 页的框图。

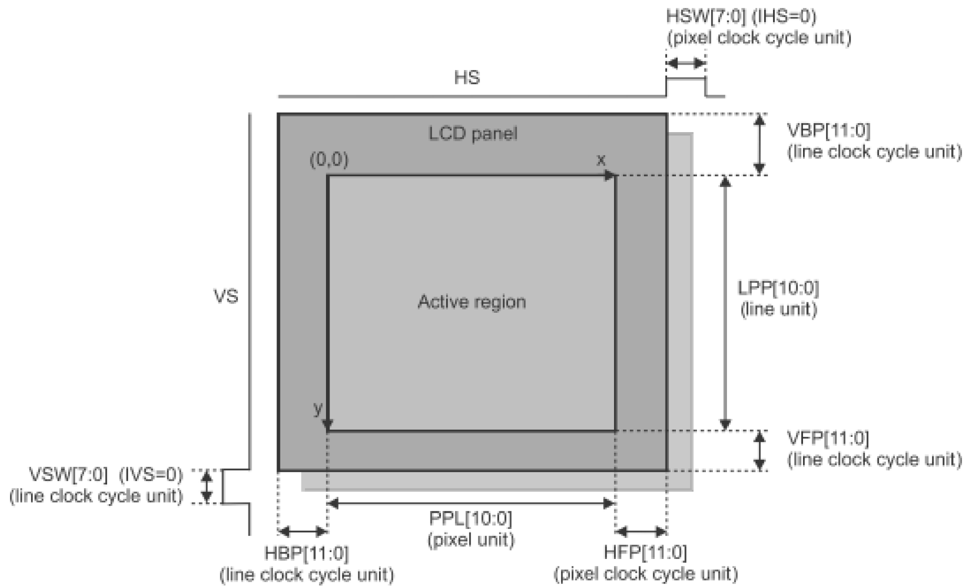


图 6-7　可变参数中硬件信息的含义

固定信息是在 frame buffer 的操作过程中并不发生变化的。具体如下：

```
struct fb_fix_screeninfo{
    char id[16];                    /* identification string eg "TT Builtin" */
    //分配内存空间的物理地址,用于驱动操作
    unsigned long smem_start;       /* Start of frame buffer mem */
                                    /* (physical address) */
    __u32 smem_len;                 /* Length of frame buffer mem */
    //设备类型,通常都是 packed pixel
    __u32 type;                     /* see FB_TYPE_* */
    __u32 type_aux;                 /* Interleave for interleaved Planes */
    //视觉效果主要是色彩数
    __u32 visual;                   /* see FB_VISUAL_* */
    //这里是进行显示内容时相对 frame buffer 虚拟分辨率的偏移最小像素单位
    __u16 xpanstep;                 /* zero if no hardware panning */
    __u16 ypanstep;                 /* zero if no hardware panning */
    __u16 ywrapstep;                /* zero if no hardware ywrap */
    __u32 line_length;              /* length of a line in bytes */
    unsigned long mmio_start;       /* Start of Memory Mapped I/O */
                                    /* (physical address) */
    __u32 mmio_len;                 /* Length of Memory Mapped I/O */
    __u32 accel;                    /* Indicate to driver which */
                                    /* specific chip/card we have */
    __u16 reserved[3];              /* Reserved for future compatibility */
};
```

其中主要的就是物理地址信息。

接下来就是驱动的重点——操作接口：

```
struct fb_ops{
    /* open/release and usage marking */
    struct module *owner;
    //打开和释放的操作
    int(*fb_open)(struct fb_info *info,int user);
    int(*fb_release)(struct fb_info *info,int user);

    /* For framebuffers with strange non linear layouts or that do not
     * work with normal memory mapped access
     */
    //特殊的内存布局使用的接口
    ssize_t(*fb_read)(struct fb_info *info,char __user *buf,
              size_t count,loff_t *ppos);
    ssize_t(*fb_write)(struct fb_info *info,const char __user *buf,
```

```
            size_t count,loff_t * ppos);

/ * checks var and eventually tweaks it to something supported,
  * DO NOT MODIFY PAR * /
//设备检查可变参数
int( * fb_check_var)(struct fb_var_screeninfo * var,struct fb_info * info);

/ * set the video mode according to info -> var * /
int( * fb_set_par)(struct fb_info * info);

/ * set color register * /
//设置设备寄存器接口
int( * fb_setcolreg)(unsigned regno,unsigned red,unsigned green,
            unsigned blue,unsigned transp,struct fb_info * info);

/ * set color registers in batch * /
//设置 color map 表
int( * fb_setcmap)(struct fb_cmap * cmap,struct fb_info * info);

/ * blank display * /
//blank/unblank 显示器
int( * fb_blank)(int blank,struct fb_info * info);

/ * pan display * /
//用于刷新显示内容
int( * fb_pan_display)(struct fb_var_screeninfo * var,struct fb_info * info);

/ * Draws a rectangle * /
//在 frame buffer 的内存空间进行 2D 的操作,可以通过设备加速
void( * fb_fillrect)(struct fb_info * info,const struct fb_fillrect * rect);
/ * Copy data from area to another * /
void( * fb_copyarea)(struct fb_info * info,const struct fb_copyarea * region);
/ * Draws a image to the display * /
void( * fb_imageblit)(struct fb_info * info,const struct fb_image * image);

/ * Draws cursor * /
int( * fb_cursor)(struct fb_info * info,struct fb_cursor * cursor);

/ * Rotates the display * /
//显示旋转
void( * fb_rotate)(struct fb_info * info,int angle);
```

```
    /* wait for blit idle,optional */
    int(*fb_sync)(struct fb_info *info);

    /* perform fb specific ioctl(optional) */
    //设备特殊的 ioctl 接口
    int(*fb_ioctl)(struct fb_info *info,unsigned int cmd,
            unsigned long arg);

    /* Handle 32bit compat ioctl(optional) */
    int(*fb_compat_ioctl)(struct fb_info *info,unsigned cmd,
            unsigned long arg);

    /* perform fb specific mmap */
    //设备特殊的 frame buffer 空间映射
    int(*fb_mmap)(struct fb_info *info,struct vm_area_struct *vma);

    /* get capability given var */
    void(*fb_get_caps)(struct fb_info *info,struct fb_blit_caps *caps,
             struct fb_var_screeninfo *var);

    /* teardown any resources to do with this framebuffer */
    void(*fb_destroy)(struct fb_info *info);

    /* called at KDB enter and leave time to prepare the console */
    int(*fb_debug_enter)(struct fb_info *info);
    int(*fb_debug_leave)(struct fb_info *info);
};
```

以上是主要的管理实体，相应的 frame buffer 框架还提供了重要的管理接口，用于驱动使用。

```
struct fb_info *framebuffer_alloc(size_t size,struct device *dev)
{
    //这里为了访问效率进行对齐 pad
#define BYTES_PER_LONG(BITS_PER_LONG/8)
#define PADDING(BYTES_PER_LONG - (sizeof(struct fb_info)% BYTES_PER_LONG))
    int fb_info_size = sizeof(struct fb_info);
    struct fb_info *info;
    char *p;

    if(size)
        fb_info_size += PADDING;
```

```
        //分配 fb_info 的空间,这里的大小是包含设备特殊信息的大小
        //形成统一空间,前面是标准的 fb_info,后面是设备特殊信息
        p = kzalloc(fb_info_size + size,GFP_KERNEL);
        if(!p)
            return NULL;

        info = (struct fb_info *)p;
        if(size)
            info -> par = p + fb_info_size;

        //初始化 fb_info 的设备关联,关联到实体化的设备
        info -> device = dev;

    #ifdef CONFIG_FB_BACKLIGHT
        mutex_init(&info -> bl_curve_mutex);
    #endif

        return info;
    #undef PADDING
    #undef BYTES_PER_LONG
    }
```

这是对 frame buffer 管理实体的分配操作，其中会初始化设备的信息。另一个重要的接口就是注册，一般是在进行相应的设置之后执行。具体内容如下：

```
    int register_framebuffer(struct fb_info * fb_info)
    {
        int i;
        struct fb_event event;
        struct fb_videomode mode;

        //注册数目已经达到最大则报错
        if(num_registered_fb == FB_MAX)
            return -ENXIO;

        if(fb_check_foreignness(fb_info))
            return -ENOSYS;

        remove_conflicting_framebuffers(fb_info -> apertures,fb_info -> fix.id,
                                fb_is_primary_device(fb_info));
```

```
    //新注册
    num_registered_fb++;
    //获得管理空间
    for(i=0 ;i<FB_MAX;i++)
        if(!registered_fb[i])
            break;
    //标记管理的 index
    fb_info->node = i;
    mutex_init(&fb_info->lock);
    mutex_init(&fb_info->mm_lock);

    //创建设备模型的设备实体,这里会包括设备号,通知应用层创建设备文件
    fb_info->dev = device_create(fb_class,fb_info->device,
                    MKDEV(FB_MAJOR,i),NULL,"fb%d",i);
    if(IS_ERR(fb_info->dev)){
        /* Not fatal */
        printk(KERN_WARNING "Unable to create device for framebuffer %d;errno = %ld\n",i,
PTR_ERR(fb_info->dev));
        fb_info->dev = NULL;
    }else
        fb_init_device(fb_info);

    //初始化 pixmap 的信息
    if(fb_info->pixmap.addr == NULL){
        fb_info->pixmap.addr = kmalloc(FBPIXMAPSIZE,GFP_KERNEL);
        if(fb_info->pixmap.addr){
            fb_info->pixmap.size = FBPIXMAPSIZE;
            fb_info->pixmap.buf_align = 1;
            fb_info->pixmap.scan_align = 1;
            fb_info->pixmap.access_align = 32;
            fb_info->pixmap.flags = FB_PIXMAP_DEFAULT;
        }
    }
    fb_info->pixmap.offset = 0;

    if(!fb_info->pixmap.blit_x)
        fb_info->pixmap.blit_x = ~(u32)0;

    if(!fb_info->pixmap.blit_y)
        fb_info->pixmap.blit_y = ~(u32)0;

    if(!fb_info->modelist.prev || !fb_info->modelist.next)
```

```
        INIT_LIST_HEAD(&fb_info->modelist);

    //根据可变信息添加标准的 mode
    fb_var_to_videomode(&mode,&fb_info->var);
    fb_add_videomode(&mode,&fb_info->modelist);
    //加入系统管理中
    registered_fb[i] = fb_info;

    //通知其他模块有 frame buffer 设备注册
    event.info = fb_info;
    if(!lock_fb_info(fb_info))
        return -ENODEV;
    fb_notifier_call_chain(FB_EVENT_FB_REGISTERED,&event);
    unlock_fb_info(fb_info);
    return 0;
}
```

这些接口是 frame buffer 框架和具体驱动交互的桥梁，通过它们，设备驱动将加入到框架中供用户使用。

从整体分析，frame buffer 的框架很直接，主要就是直接管理实际的设备，而相关的操作与应用层直接相关。相关的操作逻辑可以通过应用层的操作来理解。

6.2.3 帧缓冲应用层操作及框架适配

先以一个 DM3730 的简单例子来了解应用层如何操作 frame buffer 的。如下所示：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <string.h>
#include <linux/fb.h>
#include <linux/fb.h>
#include <asm/ioctl.h>
#include <asm/types.h>

/* IOCTL commands. */
#define OMAP_IO(num)        _IO('O',num)
```

```
//DM3730 特殊的等待垂直同步的 IO,垂直同步代表新一帧数据的开始,通过该同步
//表明硬件相关操作的安全
#define OMAPFB_WAITFORVSYNC OMAP_IO(57)
//使用 VSYNC 进行操作
#define WAIT_FOR_VSYNC      1
#define ITERATIONS 1000000000

int fd;
unsigned char * data;
struct fb_var_screeninfo var;
struct fb_fix_screeninfo fix;

//刷新 frame,有 buf_no 指定是第几个 buffer
int show_frame(int buf_no)
{
    unsigned char * buf;
    static int start;
    //以屏幕十六分之一的宽度刷一个 bar
    int width = var.yres >>4;       /* 1/16th of the screen height */
    struct fb_var_screeninfo v;
    int i =0,j =0,ret =0;
    //横条的颜色
    char color =0xAA;

    memcpy(&v,&var,sizeof(v));
    //显示内容的 y 偏移
    v.yoffset = v.yres * buf_no;
    //相应帧的起始地址
    buf = data + v.yoffset * fix.line_length;

    /* clear the frame */
    //清空整帧
    memset((void *)buf,0,fix.line_length * var.yres);

    /* draw a horizontal bar on the frame */
    //画一个水平条,start 表示起始行画 width 行
    for(i = start;i < start + width;i ++){
        for(j =0;j < var.xres * var.bits_per_pixel/8;j ++)
            buf[i * fix.line_length + j] = color;
    }
```

```
        /* pan to that frame */
        //通知驱动更新下一帧的内容
        ret = ioctl(fd,FBIOPAN_DISPLAY,&v);
        if(ret){
            printf("ioctl FBIOPAN_DISPLAY failed\n");
            return ret;
        }

        /* move the position of the bar for the next frame */
        //移动横条,如果到底部再从顶部开始
        start = start + 1;
        if(start + width > var.yres)
            start = 0;

        return 0;
}

//移动颜色横条的操作接口
int movie(int num_buf,int wait_vsync)
{
        int buf_no = 0;
        int k = 0,ret = 0;
        //迭代
        for(k = 0;k < ITERATIONS;k ++ ){
            //显示横条
            ret = show_frame(buf_no);
            if(ret)
                return ret;

            //等待 VSYNC 信号
            if(wait_vsync){
                ret = ioctl(fd,OMAPFB_WAITFORVSYNC,0);
                if(ret){
                    printf("\n ioctl OMAPFB_WAITFORVSYNC failed");
                    return ret;
                }
            }
            //表示应该操作的是下一帧的 buffer
            buf_no = ( ++ buf_no)% num_buf;
        }
        return 0;
}
```

```
//初始化操作
int init_fb( char * devname)
{
    int num_buf = 1;
    int ret = 0;

    /* Open framebuffer device */
    //打开设备
    fd = open( devname, O_RDWR);
    if( fd <= 0) {
        printf( "Could not open device\n");
    }
    //获得固定信息
    ret = ioctl( fd, FBIOGET_FSCREENINFO, &fix);
    if( ret) {
        printf( "ioctl FBIOGET_FSCREENINFO failed\n");
        return ret;
    }
    //获得可变信息
    ret = ioctl( fd, FBIOGET_VSCREENINFO, &var);
    if( ret) {
        printf( "ioctl FBIOGET_VSCREENINFO failed\n");
        return ret;
    }

    //映射整个 frame buffer 的存储空间
    data = ( unsigned char * )mmap(0,
                fix. line_length * var. yres_virtual,
                (PROT_READ | PROT_WRITE),
                MAP_SHARED, fd, 0);
    if( ! data) {
        printf( "mmap failed for %d x %d of %d\n", \
            fix. line_length, var. yres_virtual, fix. smem_len);
        return -ENOMEM;
    }
    //计算真正 buffer 能存放的帧数并返回
    num_buf = var. yres_virtual/var. yres;
    return num_buf;
}

int main( int argc, char * argv[ ])
```

```
{
    int i = 0, ret = 0;
    int num_buf;
    if( ! argv[1]) {
        printf("/dev/fbN parameter missing\n");
        return -ENODEV;
    }
    //初始化
    if((num_buf = init_fb(argv[1])) < 0) {
        printf("Could not initialize dev %s\n", argv[1]);
        return -1;
    }
    //移动横条
    ret = movie(num_buf, WAIT_FOR_VSYNC);
    close(fd);
    return ret;
}
```

从这个例子基本可以明确应用是如何使用 frame buffer 的，操作流程进行总结如图 6-8 所示。

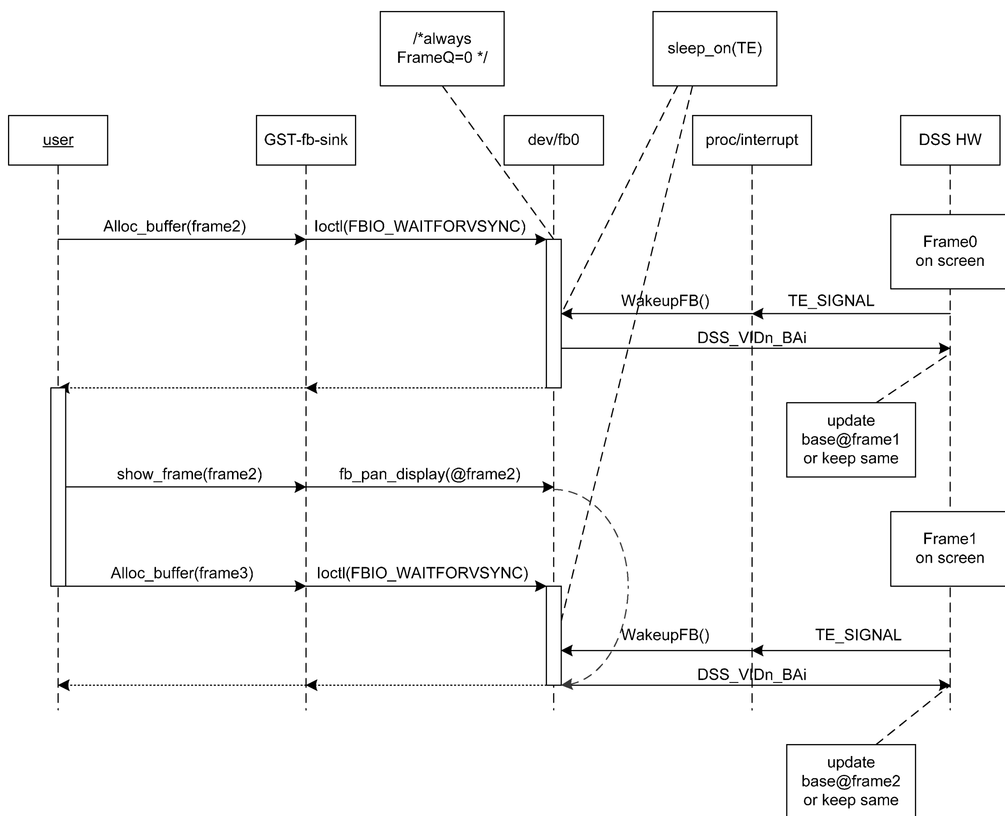


图 6-8　应用对 frame buffer 的操作流程

Android 系统中，其对应 frame buffer 的适配过程也是类似的，由于系统还要操作如 3D 加速的 Graphic 设备，所以 Android 将二者归为 gralloc 设备，frame buffer 作为其子设备存在。

Android 针对 frame buffer 的适配操作都是在 Framebuffer. cpp 中实现的。其中 mapFrameBufferLocked 负责映射空间，相关代码如下：

```
int err;
size_t fbSize = roundUpToPageSize(finfo.line_length * info.yres_virtual);
module->framebuffer = new private_handle_t(dup(fd),fbSize,0);

module->numBuffers = info.yres_virtual/info.yres;
module->bufferMask = 0;

void * vaddr = mmap(0,fbSize,PROT_READ | PROT_WRITE,MAP_SHARED,fd,0);
if(vaddr == MAP_FAILED){
ALOGE("Error mapping the framebuffer(%s)",strerror(errno));
return -errno;
}
module->framebuffer->base = intptr_t(vaddr);
memset(vaddr,0,fbSize);
```

而显示切换则由 fb_post 实现，相关代码如下：

```
if(hnd->flags & private_handle_t::PRIV_FLAGS_FRAMEBUFFER){
    const size_t offset = hnd->base - m->framebuffer->base;
    m->info.activate = FB_ACTIVATE_VBL;
    m->info.yoffset = offset/m->finfo.line_length;
    if(ioctl(m->framebuffer->fd,FBIOPAN_DISPLAY,&m->info) == -1){
        ALOGE("FBIOPAN_DISPLAY failed");
        m->base.unlock(&m->base,buffer);
        return -errno;
    }

#ifndef OMAP_FB
    if(ioctl(m->framebuffer->fd,FBIO_WAITFORVSYNC,0)){
        ALOGE("FBIO_WAITFORVSYNC failed");
        m->base.unlock(&m->base,buffer);
        return -errno;
    }
#else
    #define OMAP_IO(num)           _IO('O',num)
    #define OMAPFB_WAITFORVSYNC         OMAP_IO(57)
```

```
        unsigned int dummy;
        if(ioctl(m->framebuffer->fd,OMAPFB_WAITFORVSYNC,&dummy) <0){
            ALOGE("OMAPFB_WAITFORGO failed");
            return 0;
        }
#endif
```

Android 会对这些接口通过统一的封装提供给应用层从而完成适配。

6.2.4 TI 芯片帧缓冲驱动相关实现详解

TI DM3730 的 frame buffer 驱动针对的硬件设备是 display controller，硬件框图如图 6-9 所示。图 6-9 引自《DM3730 芯片手册》中第 1624 页框图。

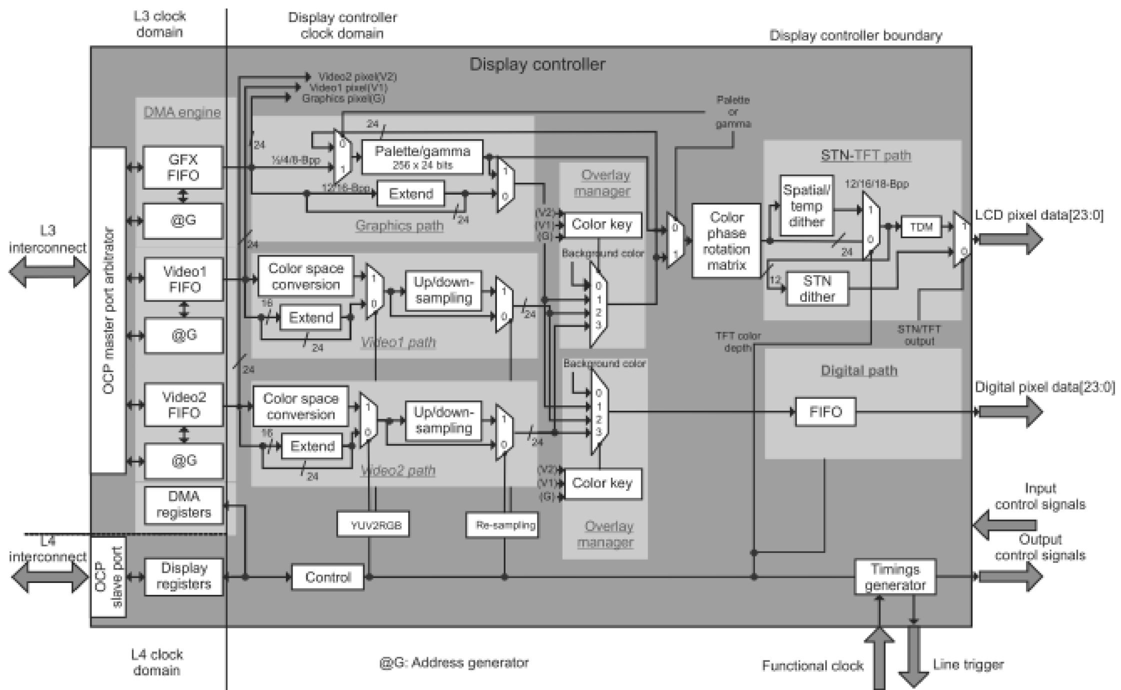


图 6-9　DM3730 display 控制器硬件框图

从图 6-9 可见，DM3730 支持 3 个不同的显示通道，而且所有的通道都通过 DMA 自动获得显示信息进行显示，不同的通道可以以一定顺序进行叠加，在显示硬件上也可以有两个显示设备，硬件提供了在不同的显示通道以及硬件显示设备切换的能力。

1. 整体框架

为了支持硬件的各种功能，在相应的驱动设计过程中 TI 设计了 DSS（display subsystem）架构支持该功能。整体的架构如图 6-10 所示。

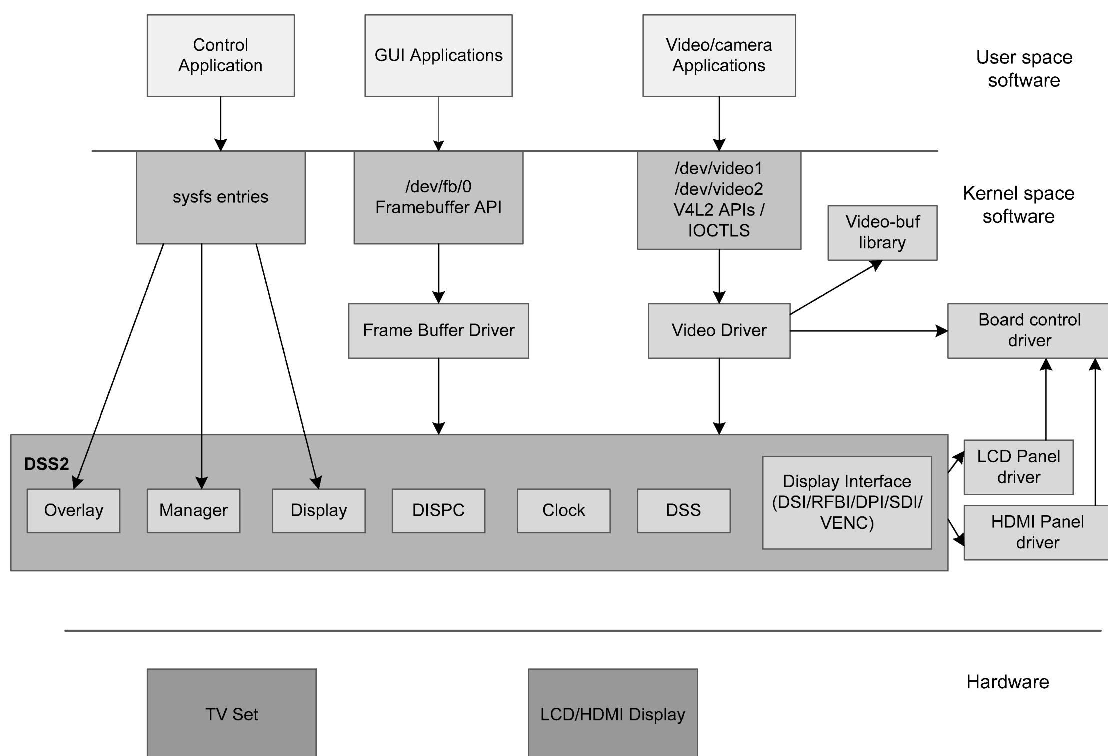


图 6-10　DM3730 显示子系统软件整体架构

从图 6-10 可见，在驱动框架部分，DSS 的设计支持 frame buffer 和 V4L2 两种架构，这是由于两种类型的设备都是通过硬件的 display 子系统进行显示，它们之间可以通过叠加子系统进行叠加，这样相当于在硬件层进行了硬件加速，可以实现良好的显示效果。

在系统设计中还开放了 sys 文件系统接口，可以方便地进行切换和设置，来满足各种动态的需求。

DSS 系统中各个模块与硬件的对应关系如图 6-11 所示。

从图 6-11 可见，不同的 pipeline 是由 overlay 进行管理来表示不同的 overlay 层，而 manager 则是管理硬件的叠加控制以及之后的显示通路，会对应到显示接口，display 部分是对显示接口的控制，panel 则是对应真实的显示设备。

开放的 sys 接口如下：

```
/sys/class/graphics/fb?：控制 frame  buffer 接入哪个 pipeline 即逻辑的 overlay 层
mirror          0 = off,1 = on
rotate          Rotation 0 – 3 for 0,90,180,270 degrees
overlays        List of overlay numbers to which framebuffer pixels go
phys_addr       Physical address of the framebuffer
virt_addr       Virtual address of the framebuffer
size            Size of the framebuffer

/sys/devices/platform/omapdss/overlay?：控制叠加层信息以及连接哪个硬件叠加管理器
```

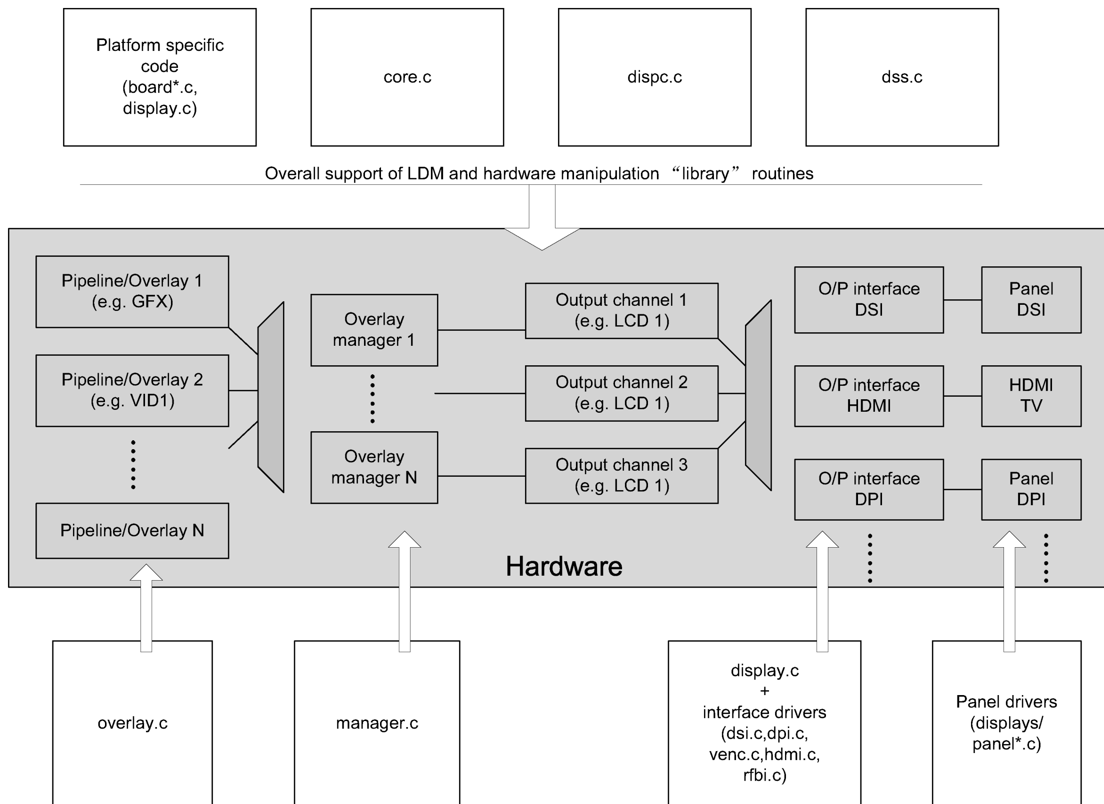


图 6-11　DSS 软件模块与硬件对应关系图

```
enabled              0 = off,1 = on
input_size           width,height(ie. the framebuffer size)
manager              Destination overlay manager name
name
output_size          width,height
position             x,y
screen_width         width
global_alpha         global alpha 0 - 255 0 = transparent     255 = opaque

/sys/devices/platform/omapdss/manager?：硬件叠加的参数以及之后显示通路的设置
display              Destination display name
alpha_blending_enabled     0 = off,1 = on
trans_key_enabled     0 = off,1 = on
trans_key_type            gfx - destination,video - source
trans_key_value           transparency color key(RGB24)
default_color     default background color(RGB24)

/sys/devices/platform/omapdss/display?：显示接口的信息和设置
ctrl_name                 Controller name
```

```
mirror              0 = off,1 = on
update_mode         0 = off,1 = auto,2 = manual
enabled             0 = off,1 = on
name
rotate              Rotation 0 - 3 for 0,90,180,270 degrees
Timings             Display timings     (pixclock,xres/hfp/hbp/hsw,yres/vfp/vbp/vsw)
tv - out:           "pal" and "ntsc"
panel_name
tear_elim           Tearing elimination 0 = off,1 = on
```

通过以上的设置就可以实现显示的切换、叠加等复杂的功能。

2. Frame Buffer 驱动

接下来看看 frame buffer 驱动是如何实现的。TI frame buffer 驱动的核心管理实体是 omapfb2_device。

```
struct omapfb2_device{
    struct device * dev;
    struct mutex mtx;

    u32 pseudo_palette[17];

    int state;

    unsigned num_fbs;
    struct fb_info * fbs[10];
    struct omapfb2_mem_region regions[10];

    unsigned num_displays;
    struct omap_dss_device * displays[10];
    unsigned num_overlays;
    struct omap_overlay * overlays[10];
    unsigned num_managers;
    struct omap_overlay_manager * managers[10];

    unsigned num_bpp_overrides;
    struct{
        struct omap_dss_device * dssdev;
        u8 bpp;
    }bpp_overrides[10];
};
```

从内容看其管理了上层 fb_info 以及内部的 overlay、manager 还有 display，实际它就是 display subsystem 的逻辑实体。

相关的初始化是在 platform 驱动的 probe 中进行的，具体分析如下：

```
//进行 omapfb 的驱动与 device 绑定的接口,主要是在系统初始化是执行
//主要是初始化驱动管理的各个接口,并创建与 kernel 之间的接口 frame  buffer
static int omapfb_probe(struct platform_device * pdev)
{
    struct omapfb2_device * fbdev = NULL;
    int r = 0;
    int i;
    struct omap_overlay * ovl;
    struct omap_dss_device * def_display;
    struct omap_dss_device * dssdev;

    //omapfb driver 不支持对于 mem、irq,dma 等 resource 的注册
    if(pdev -> num_resources != 0) {
        dev_err(&pdev -> dev,"probed for an unknown device\n");
        r = -ENODEV;
        goto err0;
    }

    //分配相应 driver 管理结构 omapfb2_device 的空间
    fbdev = kzalloc(sizeof(struct omapfb2_device),GFP_KERNEL);
    if(fbdev == NULL) {
        r = -ENOMEM;
        goto err0;
    }

    /* TODO : Replace cpu check with omap_has_vrfb once HAS_FEATURE
     *        available for OMAP2 and OMAP3
     */
    if(def_vrfb && !cpu_is_omap24xx() && !cpu_is_omap34xx()) {
        def_vrfb = 0;
        dev_warn(&pdev -> dev,"VRFB is not supported on this hardware,"
                "ignoring the module parameter vrfb = y\n");
    }

    mutex_init(&fbdev -> mtx);

    //将 omapfb2_device 加入到 kernel 的 ldm 的 device 的私有 data 中
    fbdev -> dev = &pdev -> dev;
    platform_set_drvdata(pdev,fbdev);
```

```
//由于还没有加载 display,所以先做初始化
r = 0;
fbdev -> num_displays = 0;
dssdev = NULL;
//遍历 dss bus 获得所有的 display 设备
for_each_dss_dev(dssdev) {
    //获得相应的 ldm 的 device,从而避免 device 被 release 从而被 remove,由于
    // for_each_dss_dev 中也会通过 omap_dss_get_next_device 来 put 之前的 device
    omap_dss_get_device(dssdev);

    if(!dssdev -> driver) {
        dev_err(&pdev -> dev,"no driver for display\n");
        r = -ENODEV;
    }

    //记录 omapfb2_device 管理的所有 display
    fbdev -> displays[fbdev -> num_displays ++ ] = dssdev;
}

if(r)
    goto cleanup;

if(fbdev -> num_displays == 0) {
    dev_err(&pdev -> dev,"no displays\n");
    r = -EINVAL;
    goto cleanup;
}

//获得所有的 overlay 和 overlay_manager
fbdev -> num_overlays = omap_dss_get_num_overlays();
for(i = 0;i < fbdev -> num_overlays;i ++ )
    fbdev -> overlays[i] = omap_dss_get_overlay(i);

fbdev -> num_managers = omap_dss_get_num_overlay_managers();
for(i = 0;i < fbdev -> num_managers;i ++ )
    fbdev -> managers[i] = omap_dss_get_overlay_manager(i);

//解析并设置相应的 display mode
if(def_mode && strlen(def_mode) > 0) {
    if(omapfb_parse_def_modes(fbdev))
        dev_warn(&pdev -> dev,"cannot parse default modes\n");
}
```

```
//创建 kernel 的 frame    buffer 接口,主要是分配显存初始化 fb_info 等
r = omapfb_create_framebuffers(fbdev);
if(r)
    goto cleanup;

//针对之前初始化的结果设置 dispc 在 dss2 框架下的 overlay manager
//的配置应用 apply 操作
for(i = 0;i < fbdev -> num_managers;i ++ ){
    struct omap_overlay_manager * mgr;
    mgr = fbdev -> managers[i];
    r = mgr -> apply(mgr);
    if(r)
        dev_warn(fbdev -> dev,"failed to apply dispc config\n");
}

/ * gfx overlay should be the default one. find a display
 * connected to that,and use it as default display * /
//默认的 overlay 为 graphic overlay,获得默认的显示设置 gfx
ovl = omap_dss_get_overlay(0);
if(ovl -> manager && ovl -> manager -> device){
    def_display = ovl -> manager -> device;
}else{
    dev_warn(&pdev -> dev,"cannot find default display\n");
    def_display = NULL;
}

if(def_display){
    //有默认的 display 则 enable
    struct omap_dss_driver * dssdrv = def_display -> driver;

    r = def_display -> driver -> enable(def_display);
    if(r){
        dev_warn(fbdev -> dev,"Failed to enable display '%s'\n",
                def_display -> name);
        goto cleanup;
    }

    //根据 manual 或者 auto update 方式进行必要的设置
    if(def_display -> caps & OMAP_DSS_DISPLAY_CAP_MANUAL_UPDATE){
        u16 w,h;
        if(dssdrv -> enable_te)
```

```
                        dssdrv->enable_te(def_display,1);
                    if(dssdrv->set_update_mode)
                     dssdrv->set_update_mode(def_display,OMAP_DSS_UPDATE_MANUAL);

                    dssdrv->get_resolution(def_display,&w,&h);
                    def_display->driver->update(def_display,0,0,w,h);
                }else{
                    if(dssdrv->set_update_mode)
                     dssdrv->set_update_mode(def_display,OMAP_DSS_UPDATE_AUTO);
                }
            }

            DBG("create sysfs for fbs\n");
            //创建 sysfs 为设置开放相应的接口
            r = omapfb_create_sysfs(fbdev);
            if(r){
                dev_err(fbdev->dev,"failed to create sysfs entries\n");
                goto cleanup;
            }

            return 0;

        cleanup:
            omapfb_free_resources(fbdev);
        err0:
            dev_err(&pdev->dev,"failed to setup omapfb\n");
            return r;
        }
```

通过以上初始化后，就可以通过 sys 文件系统对相关属性进行设置了。另外 frame buffer 的可变参数也是可以修改的（如 FBIOPAN_DISPLAY 命令）。驱动应对这种修改通过 mapfb_apply_changes 来实现。具体分析如下：

```
        //omap 设置对 frame  buffer 的 overlay 的修改。该函数会在标准的 fram  buffer ioctl
        //omap 特殊的 ioctl 以及 sysfs 等需要进行 frame  buffer 到 overlay 切换的场景调用
        int omapfb_apply_changes(struct fb_info *fbi,int init)
        {
            int r = 0;
            struct omapfb_info *ofbi = FB2OFB(fbi);
            struct fb_var_screeninfo *var = &fbi->var;
            struct omap_overlay *ovl;
            u16 posx,posy;
```

```
	u16 outw,outh;
	int i;

#ifdef DEBUG
	if(omapfb_test_pattern)
		fill_fb(fbi);
#endif
	WARN_ON(!atomic_read(&ofbi->region->lock_count));

	//对 frame  buffer 绑定的 overlay 都进行设置
	for(i=0;i<ofbi->num_overlays;i++){
		ovl = ofbi->overlays[i];

		if(ofbi->region->size==0){
			/* the fb is not available. disable the overlay */
			//对于没有分配 frame  buffer 的,disable overlay
			omapfb_overlay_enable(ovl,0);
			if(!init && ovl->manager)
				ovl->manager->apply(ovl->manager);
			continue;
		}

		//根据 overlay 的属性设置输出的分辨率
		if(init || (ovl->caps & OMAP_DSS_OVL_CAP_SCALE)==0){
			//根据旋转情况设置输出分辨率
			int rotation = (var->rotate + ofbi->rotation[i])% 4;
			if(rotation == FB_ROTATE_CW ||
					rotation == FB_ROTATE_CCW){
				outw = var->yres;
				outh = var->xres;
			}else{
				outw = var->xres;
				outh = var->yres;
			}
		}else{
			//如果 overlay 支持 scale 则设置缩放的分辨率
			outw = ovl->info.out_width;
			outh = ovl->info.out_height;
		}

		//设置显示的起始位置
		if(init){
```

```
                posx = 0;
                posy = 0;
            }else{
                posx = ovl -> info.pos_x;
                posy = ovl -> info.pos_y;
            }

            //针对 fb_info 和显示参数设置 overlay
            r = omapfb_setup_overlay(fbi,ovl,posx,posy,outw,outh);
            if(r)
                goto err;

            //apply overlay 的设置,通过 manager 显示,这里会调用 go_lcd 来刷新 shadow 寄存器
            if(!init && ovl -> manager)
                ovl -> manager -> apply(ovl -> manager);
        }
        return 0;
err:
    DBG("apply_changes failed\n");
    return r;
}
```

frame buffer 的驱动还有重要的内容就是对显示存储空间的分配和映射，下面进行相关的分析。

显示存储空间的分配系统会在初始化的时候通过启动参数预留，相关代码分析如下：

```
//用于设置 bootargs 中的 vram 设置
static int __init omap_vram_early_vram(char * p)
{
    omap_vram_def_sdram_size = memparse(p,&p);
    if(*p == ',')
        omap_vram_def_sdram_start = simple_strtoul(p + 1,&p,16);
    return 0;
}
early_param("vram",omap_vram_early_vram);

/*
 * Called from map_io. We need to call to this early enough so that we
 * can reserve the fixed SDRAM regions before VM could get hold of them.
 */
//该函数会在 machine_desc 的 reserve 接口调用,尽早的调用以保留相应的连续的 memory
//空间,所以需要在 VM 管理相应的空间之前调用
```

```
void __init omap_vram_reserve_sdram_memblock(void)
{
    u32 paddr;
    u32 size = 0;

    /* cmdline arg overrides the board file definition */
    //bootargs 的配置优先级要高,先用 bootargs 中的 vram 设置
    if(omap_vram_def_sdram_size){
        size = omap_vram_def_sdram_size;
        paddr = omap_vram_def_sdram_start;
    }

    if(!size){
        //如果 bootargs 中没有设置 vram,则用 board 修改
        size = omap_vram_sdram_size;
        paddr = omap_vram_sdram_start;
    }

#ifdef CONFIG_OMAP2_VRAM_SIZE
    if(!size){
        //没有设置的话则用默认的设置
        //默认 4M
        size = CONFIG_OMAP2_VRAM_SIZE * 1024 * 1024;
        paddr = 0;
    }
#endif

    if(!size)
        return;

    //size 为 2M 对齐
    size = ALIGN(size,SZ_2M);

    if(paddr){
        //如果指定起始地址则进行检查
        //页对齐检查
        if(paddr & ~PAGE_MASK){
            pr_err("VRAM start address 0x%08x not page aligned\n",
                    paddr);
            return;
        }
```

```
            //地址范围是否在 kernel 管理的 region 中
            if(!memblock_is_region_memory(paddr,size)){
                pr_err("Illegal SDRAM region 0x%08x..0x%08x for VRAM\n",
                        paddr,paddr + size - 1);
                return;
            }

            //是否已经 reserve
            if(memblock_is_region_reserved(paddr,size)){
                pr_err("FB: failed to reserve VRAM - busy\n");
                return;
            }

            //reserve 相应的空间
            if(memblock_reserve(paddr,size) < 0){
                pr_err("FB: failed to reserve VRAM - no memory\n");
                return;
            }
        }else{
            //没有指定起始地址,则直接分配相应大小空间
            //2M 对齐,返回分配的起始地址
            paddr = memblock_alloc(size,SZ_2M);
        }

        //将相应的空间从 kernel 管理的 memory block 中移除
        memblock_free(paddr,size);
        memblock_remove(paddr,size);

        //相应的空间由 vram 管理
        omap_vram_add_region(paddr,size);

        pr_info("Reserving %u bytes SDRAM for VRAM\n",size);
    }
```

显存的空间通过 vram 统一管理，这部分空间不在内核的管理范围内，相应的分配也是由 vram 的分配函数 omap_vram_alloc 来进行，具体的分配工作则是在驱动 probe 中通过 omapfb_create_framebuffers 来执行的。

驱动也提供了相应的映射接口 omapfb_ mmap，分析如下：

```
    //frame  buffer 设备驱动的 mmap 接口函数
    static int omapfb_mmap(struct fb_info * fbi,struct vm_area_struct * vma)
    {
```

```
struct omapfb_info * ofbi = FB2OFB(fbi);
struct fb_var_screeninfo * var = &fbi -> var;
struct fb_fix_screeninfo * fix = &fbi -> fix;
struct omapfb2_mem_region * rg;
unsigned long off;
unsigned long start;
u32 len;
int r = -EINVAL;

//对 vm_area 的区域进行检查
if(vma -> vm_end - vma -> vm_start == 0)
    return 0;
if(vma -> vm_pgoff > (~0UL >> PAGE_SHIFT))
    return -EINVAL;
off = vma -> vm_pgoff << PAGE_SHIFT;

//锁保护
rg = omapfb_get_mem_region(ofbi -> region);
//获得相应 omapfb 管理的物理区域的物理起始地址
start = omapfb_get_region_paddr(ofbi,var -> rotate);
//memory 的大小
len = fix -> smem_len;

//进行虚拟空间 vm_area 和物理分配显存空间的容量检查,
//不应该超出 frame    buffer 管理的 memory 空间
if(off >= len)
    goto error;
if((vma -> vm_end - vma -> vm_start + off) > len)
    goto error;

off += start;

//设置 vm_area 相关参数
vma -> vm_pgoff = off >> PAGE_SHIFT;
vma -> vm_flags |= VM_IO | VM_RESERVED;
//见 ARM MMU cache and buffer 说明,这里设置 buffered 数据
vma -> vm_page_prot = pgprot_writecombine(vma -> vm_page_prot);
//设置 vm_ops 操作接口
vma -> vm_ops = &mmap_user_ops;
//设置操作接口需要的特定 private_data
vma -> vm_private_data = rg;
```

```
        //进行实际的物理地址的映射
        if( io_remap_pfn_range( vma, vma -> vm_start, off >> PAGE_SHIFT,
                        vma -> vm_end - vma -> vm_start,
                        vma -> vm_page_prot) ) {
            r = -EAGAIN;
            goto error;
        }

        /* vm_ops. open won't be called for mmap itself. */
        //增加计数,由于 mmap 本身不会调用 vm_ops. open,这里实际
        //region 已经使用所以主动增加计数
        atomic_inc( &rg -> map_count) ;

        omapfb_put_mem_region( rg) ;

        return 0;

    error:
        omapfb_put_mem_region( ofbi -> region) ;

        return r;
    }
```

从分析中可见，设置了虚拟地址区域的操作接口 mmap_user_ops，由于相应的映射是通过 io_remap_pfn_range 的整体映射，并不需要进行缺页异常的操作，所以 mmap_user_ops 只提供了 open 和 release 的接口用于引用计数的操作。

用户对 frame buffer 的操作基本都通过以上的功能完成。

3. 显示设备的管理

从 DSS 硬件可见，其可以支持多个显示设备，这有些类似于总线，为了能够同时支持多个显示设备，驱动设计并实现了一种逻辑总线——dss_bus。相关分析如下：

```
    //bus 类型,会注册到 kernel 中的 bus
    static struct bus_type dss_bus_type = {
        . name = "omapdss",
        . match = dss_bus_match,
        . dev_attrs = default_dev_attrs,
        . drv_attrs = default_drv_attrs,
    } ;

    //设备和驱动进行 match 的接口函数
    static int dss_bus_match( struct device * dev, struct device_driver * driver)
    {
```

```
    struct omap_dss_device * dssdev = to_dss_device(dev);

    DSSDBG("bus_match. dev %s/%s,drv %s\n",
            dev_name(dev),dssdev->driver_name,driver->name);

    //通过 name 进行匹配
    return strcmp(dssdev->driver_name,driver->name) == 0;
}
```

另外在设备与驱动绑定需要相应的注册接口，具体的分析如下：

```
//dss bus 注册 device 的接口函数,目前只有使用 omap_dss_probe 来注册 board 文件
//声明的所有 dss_device
int omap_dss_register_device(struct omap_dss_device * dssdev)
{
    static int dev_num;

    //clear device 的数据
    reset_device(&dssdev->dev,1);
    //设置默认的 bus 和 parent device
    dssdev->dev.bus = &dss_bus_type;
    dssdev->dev.parent = &dss_bus;
    dssdev->dev.release = omap_dss_dev_release;
    //设置 name
    dev_set_name(&dssdev->dev,"display%d",dev_num++);
    //调用 ldm 的 device 注册接口
    return device_register(&dssdev->dev);
}

//dss bus 注册 drvier
int omap_dss_register_driver(struct omap_dss_driver * dssdriver)
{
    //首先注册总线级别的操作和 bus_type
    dssdriver->driver.bus = &dss_bus_type;
    dssdriver->driver.probe = dss_driver_probe;
    dssdriver->driver.remove = dss_driver_remove;

    if(dssdriver->get_resolution == NULL)
        dssdriver->get_resolution = omapdss_default_get_resolution;
    if(dssdriver->get_recommended_bpp == NULL)
        dssdriver->get_recommended_bpp =
            omapdss_default_get_recommended_bpp;
```

```
    //调用 ldm 的 driver 注册接口,由于 bus 默认会自动匹配所以会进行 match 操作
    //进而执行 bus 级别的 probe 即 dss_driver_probe
    return driver_register(&dssdriver->driver);
}
```

可见 dss bus 管理的实体就是设备 omap_dss_device 和驱动 omap_dss_driver，在 omap_dss_device 中主要是显示设备的硬件信号信息，而 omap_dss_driver 则是操作接口（包括功能和电源管理操作）。

如果整个 dss 编进内核的话，相应的初始化如下：

```
static int __init omap_dss_init(void)
{
    return omap_dss_bus_register();
}

static int __init omap_dss_init2(void)
{
    return platform_driver_register(&omap_dss_driver);
}

core_initcall(omap_dss_init);
device_initcall(omap_dss_init2);
```

可见 bus 的初始化要早于相应的驱动，这样就在 dss 驱动初始化 probe 的过程中将所有的显示设备信息注册，而驱动注册的时候就可以将二者绑定。

这样就完成了多种显示设备的管理，从而可以实现各种复杂的功能。

6.2.5 帧缓冲驱动电源管理相关说明

通常显示设备的电源管理功能就是通过 blank 接口实现的，先来看看 frame buffer 提供的标准接口的实现

```
int fb_blank(struct fb_info *info,int blank)
{
    int ret = -EINVAL;

    if(blank > FB_BLANK_POWERDOWN)
        blank = FB_BLANK_POWERDOWN;

    if(info->fbops->fb_blank)
        ret = info->fbops->fb_blank(blank,info);
```

```
	if(!ret){
		struct fb_event event;

		event.info = info;
		event.data = &blank;
		fb_notifier_call_chain(FB_EVENT_BLANK,&event);
	}

	return ret;
}
```

其中直接调用了 frame buffer 驱动的 blank 接口，接下来是 DM3730 的驱动实现：

```
//blank display 的接口,主要用作 blank/unblank 显示 display
static int omapfb_blank(int blank,struct fb_info * fbi)
{
	//首先要获得 display 的属性信息
	struct omapfb_info * ofbi = FB2OFB(fbi);
	struct omapfb2_device * fbdev = ofbi->fbdev;
	struct omap_dss_device * display = fb2display(fbi);
	int do_update = 0;
	int r = 0;

	if(!display)
		return -EINVAL;

	omapfb_lock(fbdev);

	switch(blank){
	case FB_BLANK_UNBLANK:
		//ublank 则 display 的状态应该是 suspend
		if(display->state != OMAP_DSS_DISPLAY_SUSPENDED)
			goto exit;

		//调用 display 的 resume 接口恢复
		if(display->driver->resume)
			r = display->driver->resume(display);

		//有些 display 在 resume 需要手动 update 用以恢复,
		//这样的 display panel 驱动会将 update mode 设为
		//OMAP_DSS_UPDATE_MANUAL,这里置标识,后面 update
```

```
            if(r == 0 && display -> driver -> get_update_mode &&
                    display -> driver -> get_update_mode(display) ==
                    OMAP_DSS_UPDATE_MANUAL)
                do_update = 1;

            break;

        case FB_BLANK_NORMAL:
            /* FB_BLANK_NORMAL could be implemented.
             * Needs DSS additions. */
        case FB_BLANK_VSYNC_SUSPEND:
        case FB_BLANK_HSYNC_SUSPEND:
        case FB_BLANK_POWERDOWN:
            //其他属于 blank 操作,检查状态应为 active
            if(display -> state != OMAP_DSS_DISPLAY_ACTIVE)
                goto exit;

            //进行 suspend 操作
            if(display -> driver -> suspend)
                r = display -> driver -> suspend(display);

            break;

        default:
            r = -EINVAL;
        }

    exit:
        omapfb_unlock(fbdev);

        //手动 update 需要进行操作
        if(r == 0 && do_update && display -> driver -> update) {
            u16 w,h;
            display -> driver -> get_resolution(display,&w,&h);

            r = display -> driver -> update(display,0,0,w,h);
        }

        return r;
    }
```

可见会找到所有 frame buffer 相对应的 omap_dss_device 即 display，然后对 display 做正确

的操作。

以上是单个显示设备的电源管理工作，而当整个系统进入待机状态时要对所有的显示设备进行相关的操作。这些操作是通过 platform driver omap_dss_driver 来实现的。

```
static struct platform_driver omap_dss_driver = {
	.probe		= omap_dss_probe,
	.remove		= omap_dss_remove,
	.shutdown	= omap_dss_shutdown,
	.suspend	= omap_dss_suspend,
	.resume		= omap_dss_resume,
	.driver		= {
		.name	= "omapdss",
		.owner	= THIS_MODULE,
	},
};
```

具体看一下 suspend 操作：

```
//suspend 操作接口
static int omap_dss_suspend(struct platform_device *pdev,pm_message_t state)
{
	//遍历 dss bus 的所有设备进行 suspend 操作
	return dss_suspend_all_devices();
}
```

这样会 suspend 所有的 device，具体通过 dss_suspend_device 对每个 display 进行 suspend 操作，细节如下：

```
//将 display device suspend
static int dss_suspend_device(struct device *dev,void *data)
{
	int r;
	struct omap_dss_device *dssdev = to_dss_device(dev);

	//如果不是 active 状态,则需要标记,resume 时不应该恢复
	if(dssdev->state != OMAP_DSS_DISPLAY_ACTIVE){
		dssdev->activate_after_resume = false;
		return 0;
	}

	if(!dssdev->driver->suspend){
		DSSERR("display '%s' doesn't implement suspend\n",
				dssdev->name);
```

```
            return - ENOSYS;
        }

        //调用 display device driver 的 suspend
        r = dssdev -> driver -> suspend(dssdev);
        if(r)
            return r;

        //设置 resume 后需要恢复的状态
        dssdev -> activate_after_resume = true;

        return 0;
    }
```

这样就实现了单个设备级别以及整个系统级别在 frame buffer 层面的电源管理功能。

6.3　音频设备（audio ALSA）

6.3.1　音频设备需求

对计算机以及嵌入式设备来说，音频设备同样是一种重要的输出设备，由于人们对声音是十分敏感的，所以在有人机交互功能的设备中音频设备的重要性就不言而喻了。

声音是连续的模拟信号，而音频设备是要将这些模拟信号与数字信号进行转换，从而满足音频的输入输出的需求。而这种模拟信号和数字信号转换的原理如图 6-12 所示。

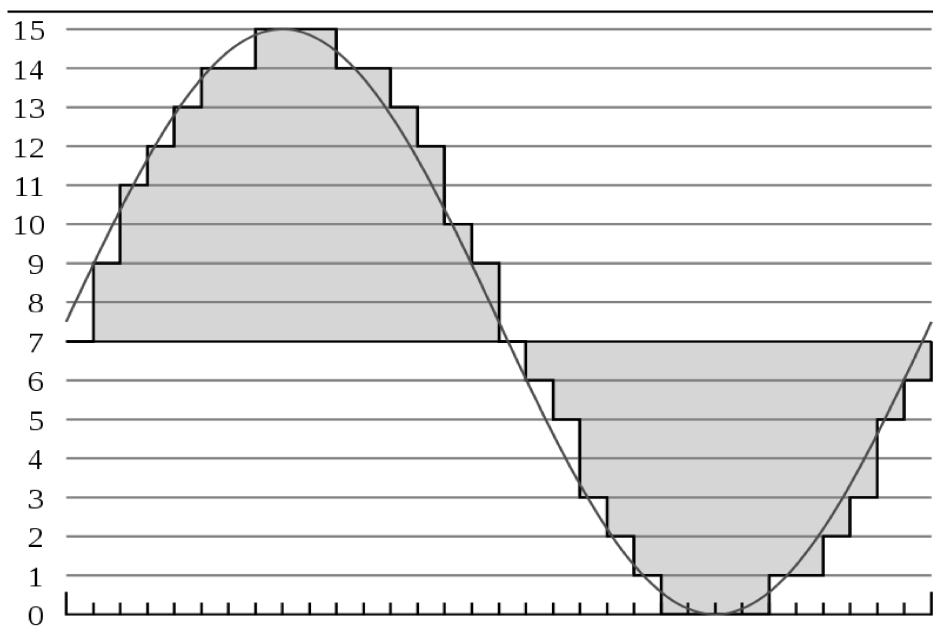


图 6-12　音频信号模数转换的原理

从图 6-11 可见这种转换的基本原理就是通过离散时间对信号进行量化来实现对连续模拟信号尽量准确地模拟。根据这个原理可知对音频信号数字效果的影响因素如下：

- 采样频率。采样频率是指单位时间内的采样次数。采样频率越大，采样点之间的间隔就越小，数字化后得到的声音就越逼真，但相应的数据量就越大。声音采样频率以 kHz（千赫兹）衡量。
- 量化位数（采样位数）。量化位数是模拟量转换成数字量之后的数据位数。量化位数表示的是声音的振幅，位数越多，音质越细腻，相应的数据量就越大。量化位数主要有 8 位和 16 位两种。
- 声道数。声道数是指处理的声音是单声道还是立体声。单声道在声音处理过程中只有单数据流，而立体声则需要左、右声道的两个数据流。显然，立体声的效果要好，但相应的数据量要比单声道的数据量加倍。

不同的音频效果就是以上因素的各种组合，而音频设备与主处理器之间的实际音频流就是通过以上因素获得的数字音频信息。所以驱动要能够支持各种组合的音频信号，其中必然包含数据流的传输支持以及控制支持。另外音频通常包括输入和输出，输入和输出是允许同时进行的，所以还要能够支持音频流输入输出的同时传送。

另外还有很多产生和制作数字音频的技术（如 Sequencer、MIDI），也是需要与音频设备进行相应的支持，这属于可选功能。

除了以上数字音频的部分（与主处理器关联的都是数字信号），音频设备还有模拟部分，这部分对于驱动的需求通常是在控制上，数据流都是转换成数字信号进行存储。这样在需求上就需要分为控制部分和数据部分，驱动也要对这些进行支持。

音频设备还有可能有多路数据源，而在设备内部作混音，所以在考虑控制需求的时候，同样要考虑相关的应用以及混音通路的设置。这样更进一步要求控制流能够和数据流分离。然而控制流和数据流不仅要考虑分离又要考虑关联，毕竟控制部分同样需要能够获取数据流格式等相关的信息。

总体来说，音频设备可理解为有单一控制通路，有一个或多个数据通路的设备。

6.3.2 音频驱动框架解析

目前 Linux 内核的音频框架采用 ALSA（Advance Linux Sound Architecture）架构，该架构的设计充分考虑了以上各种需求。ALSA 整体框架如图 6-13 所示。

从图 6-13 可见 ALSA 不仅包括驱动部分，还包括应用层的库。应用层的库主要是对驱动提供的服务进行封装，可以避免应用程序直接进行一些繁杂的 ioctl 调用，通过库的封装可以使得调用流程更易于理解，从而整体更易于应用程序的开发。

1. 整体设备管理

这里主要对于 ALSA 架构的驱动部分进行解析。首先从整个框架上看，ALSA 可以分为 control、pcm、midi、sequencer 等几个部分，对这些部分通常是通过多个设备文件来实现应用与内核交互的。框架层是如何对设备文件进行划分的呢？这一点可以通过子设备号的划分来了解。ALSA 提供子设备号的静态分配和动态分配两种方式。通过静态分配更容易了解各个设备文件的特点。下面来看看相关代码：

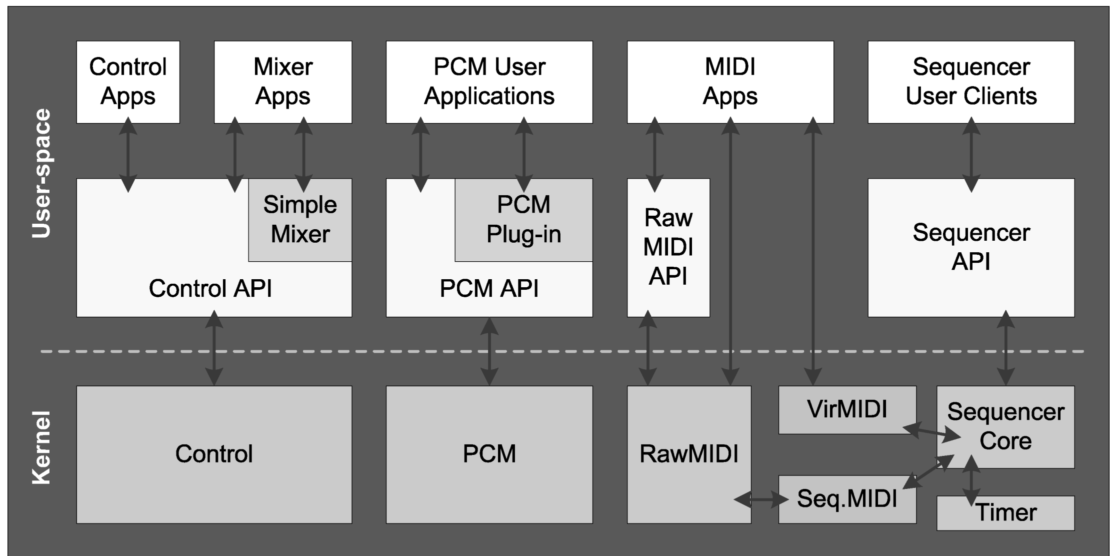


图 6-13　ALSA 整体框架

```
static int snd_kernel_minor(int type,struct snd_card * card,int dev)
{
    int minor;

    switch(type) {
    case SNDRV_DEVICE_TYPE_SEQUENCER:
    case SNDRV_DEVICE_TYPE_TIMER:
        minor = type;
        break;
    case SNDRV_DEVICE_TYPE_CONTROL:
        if(snd_BUG_ON(!card))
            return -EINVAL;
        minor = SNDRV_MINOR(card->number,type);
        break;
    case SNDRV_DEVICE_TYPE_HWDEP:
    case SNDRV_DEVICE_TYPE_RAWMIDI:
    case SNDRV_DEVICE_TYPE_PCM_PLAYBACK:
    case SNDRV_DEVICE_TYPE_PCM_CAPTURE:
        if(snd_BUG_ON(!card))
            return -EINVAL;
        minor = SNDRV_MINOR(card->number,type + dev);
        break;
    default:
        return -EINVAL;
    }
```

```
        if(snd_BUG_ON(minor < 0 || minor >= SNDRV_OS_MINORS))
            return -EINVAL;
        return minor;
    }
```

从代码中可以了解不同的设备类型以及设备特点。从分类的角度，音频设备中可以包括的类型有 SEQUENCER、TIMER、CONTROL、HWDEP、RAWMIDI、PCM_PLAYBACK 以及 PCM_CAPTURE。从代码可见，除了 SEQUENCER 和 TIMER 之外，都是和 snd_card 相关联的。这是由于 SEQUENCER 和 TIMER 要用于系统级的音频设备，SEQUENCER 允许使用系统中所有的 MIDI 设备，而 TIMER 是允许获得系统中所有音频设备中的 timer，自然是系统级的设备，就不需要与 card 进行关联。而其他类型的设备都是与 snd_card 有关联的，其中 CONTROL 每个 snd_card 只能有一个，其他的设备则是允许有多个。这种数量的关系一定要明确。

对众多的音频设备来说，CONTROL、PCM_PLAYBACK 和 PCM_CAPTURE 是一定要存在的，其他的设备根据具体情况可有可无。所以后续主要以 CONTROL、PCM_PLAYBACK 和 PCM_CAPTURE 进行分析。

从以上部分可见，ALSA 对设备管理是有层次的，组织上 snd_card 是设备的容器，同时也是设备的组织者与管理者，逻辑上 snd_card 是与硬件声卡关联。声卡硬件本身包含不同功能的硬件模块，这些不同的模块逻辑上也是设备，所以软件的这种组织与硬件上的逻辑组织形式是一致的。对驱动来说，需要实现的是整个声卡的驱动，这样在组织上要以 card 为核心，驱动中功能性的操作则要能够深入到具体对应的子模块。

要了解 ALSA 如何组织这些不同层次的设备，先来看看 snd_card 的内容：

```
struct snd_card{
    //表示系统中的 card 号,每个 card 唯一
    int number;                          /* number of soundcard(index to snd_cards) */
    //方便理解的各种名字
    char id[16];                         /* id string of this card */
    char driver[16];                     /* driver name */
    char shortname[32];                  /* short name of this soundcard */
    char longname[80];                   /* name of this soundcard */
    char mixername[80];                  /* mixer name */
    char components[128];                /* card components delimited with space */
    struct module * module;              /* top - level module */

    //用于实例化指向具体的管理实体
    void * private_data;                 /* private data for soundcard */
    void( * private_free)(struct snd_card * card);  /* callback for freeing of private data */
    //这里是 card 包含的具体设备列表,通过链表连接起来
    struct list_head devices;            /* devices */
```

```
    unsigned int last_numid;                    /* last used numeric ID */
    struct rw_semaphore controls_rwsem;         /* controls list lock */
    rwlock_t ctl_files_rwlock;                  /* ctl_files list lock */
    //控制接口,整个 card 统一的控制接口
    int controls_count;                         /* count of all controls */
    int user_ctl_count;                         /* count of all user controls */
    struct list_head controls;                  /* all controls for this card */
    struct list_head ctl_files;                 /* active control files */

    //proc 相关的接口
    struct snd_info_entry *proc_root;           /* root for soundcard specific files */
    struct snd_info_entry *proc_id;             /* the card id */
    struct proc_dir_entry *proc_root_link;      /* number link to real id */

    //关联到该 card 上所有打开的文件,设备特殊情况下需要对文件进行一定操作
    struct list_head files_list;                /* all files associated to this card */
    struct snd_shutdown_f_ops *s_f_ops;         /* file operations in the shutdown state */
    spinlock_t files_lock;                      /* lock the files for this card */
    int shutdown;                               /* this card is going down */
    int free_on_last_close;                     /* free in context of file_release */
    wait_queue_head_t shutdown_sleep;
    //这里指向该 card 在设备模型中的 parent 设备
    struct device *dev;                         /* device assigned to this card */
    //设备模型中 card 的设备实体
    struct device *card_dev;                    /* cardX object for sysfs */

#ifdef CONFIG_PM
//电源管理相关部分
    unsigned int power_state;                   /* power state */
    struct mutex power_lock;                    /* power lock */
    wait_queue_head_t power_sleep;
#endif
…
};
```

从 snd_card 的整体分析可见，其主要负责整体的功能以及设备的组织。对于整体功能主要是控制功能以及整体的电源管理方面；在设备组织方面，是通过 devices 链表来对所有的子设备进行管理，子设备的类型就是之前介绍的各种类型。为了方便管理，ALSA 将各种类型的子设备统一抽象出 snd_device，用于 card 对子设备进行管理，细节如下：

```
struct snd_device{
    //进行设备连接
```

```
        struct list_head list;                    /* list of registered devices */
        //关联的 card 实体
        struct snd_card * card;                   /* card which holds this device */
        //设备所处的状态与设备是否注册,还是只加入到 card 进行管理
        snd_device_state_t state;                 /* state of the device */
        //设备类型
        snd_device_type_t type;                   /* device type */
        //设备的实例化信息,各种类型设备不同
        void * device_data;                       /* device structure */
        //实例化的操作
        struct snd_device_ops * ops;              /* operations */
    };
```

可见主要进行不同类型子设备实例化的部分就是 type、device_data 以及 ops。其中 snd_device_ops 的定义如下：

```
    struct snd_device_ops{
        int( * dev_free)(struct snd_device * dev);
        int( * dev_register)(struct snd_device * dev);
        int( * dev_disconnect)(struct snd_device * dev);
    };
```

主要是操作接口，这样可以将操作与属性完全隔离，方便后续扩展。

系统还提供统一实例化各种类型设备的接口 snd_device_new，内容如下：

```
    int snd_device_new(struct snd_card * card,snd_device_type_t type,
                       void * device_data,struct snd_device_ops * ops)
    {
        struct snd_device * dev;

        if(snd_BUG_ON(!card || !device_data || !ops))
            return -ENXIO;
        //分配 snd_device 所需空间
        dev = kzalloc(sizeof( * dev),GFP_KERNEL);
        if(dev == NULL){
            snd_printk(KERN_ERR "Cannot allocate device\n");
            return -ENOMEM;
        }
        //初始化 snd_device 信息,通过参数初始化设备实例化信息
        dev -> card = card;
        dev -> type = type;
        //表明设备状态为刚建立
        dev -> state = SNDRV_DEV_BUILD;
```

```
    dev -> device_data = device_data;
    dev -> ops = ops;
    //加入 card 的管理设备链表中
    list_add(&dev -> list,&card -> devices);      /* add to the head of list */
    return 0;
}
```

每种类型的子设备都是通过 snd_device_new 接口来创建设备并与 card 进行关联的。以 control 子设备为例，control 子设备是通过 snd_ctl_create 创建的。

```
int snd_ctl_create(struct snd_card * card)
{
    static struct snd_device_ops ops = {
        .dev_free = snd_ctl_dev_free,
        .dev_register =      snd_ctl_dev_register,
        .dev_disconnect = snd_ctl_dev_disconnect,
    };

    if(snd_BUG_ON(!card))
        return -ENXIO;
    return snd_device_new(card,SNDRV_DEV_CONTROL,card,&ops);
}
```

可见 control 子设备是和 card 直接关联的，也说明了具体的控制由 card 统一来执行。最终 card 以及子设备的关联关系如图 6-14 所示。

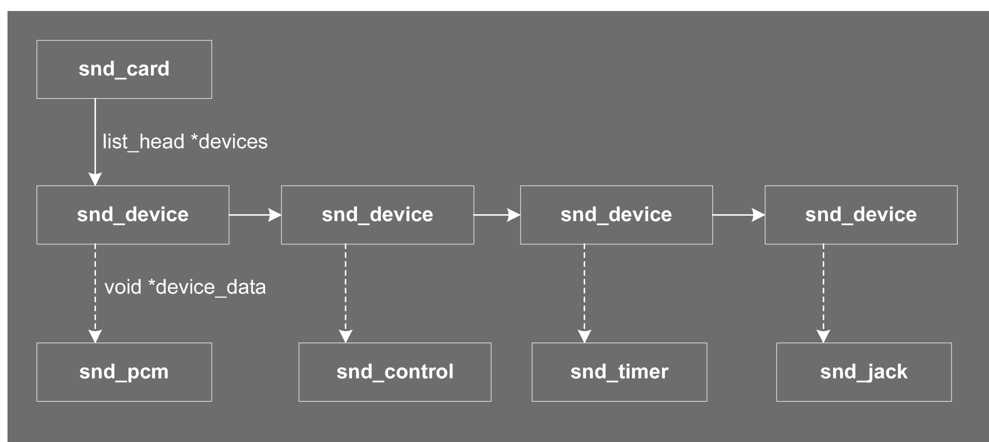


图 6-14　snd_card 及子设备关联关系

以上主要是设备组织的部分，与设备模型的注册以及设备文件的产生并没有关联。思考一下，这些工作应该由子设备自己来完成，并进行文件操作的设定，这个操作就是由 snd_device_ops 中的 dev_register 操作完成的。相应的 ALSA 为每种类型的子设备提供了相关的接

□ snd_register_device_for_dev。内容分析如下：

```
int snd_register_device_for_dev(int type,struct snd_card * card,int dev,
                                const struct file_operations * f_ops,
                                void * private_data,
                                const char * name,struct device * device)
{
    int minor;
    struct snd_minor * preg;

    if(snd_BUG_ON(!name))
        return -EINVAL;
    //分配 snd_minor 的空间,用于运行时对系统内设备号及对应操作的管理
    preg = kmalloc(sizeof * preg,GFP_KERNEL);
    if(preg == NULL)
        return -ENOMEM;
    //按照设备类型等传入参数初始化 snd_minor
    preg -> type = type;
    preg -> card = card ? card -> number : -1;
    preg -> device = dev;
    preg -> f_ops = f_ops;
    preg -> private_data = private_data;
    mutex_lock(&sound_mutex);
    //根据设备类型获得子设备号
#ifdef CONFIG_SND_DYNAMIC_MINORS
    minor = snd_find_free_minor();
#else
    //静态方式获得
    minor = snd_kernel_minor(type,card,dev);
    if(minor >= 0 && snd_minors[minor])
        minor = -EBUSY;
#endif
    if(minor < 0){
        mutex_unlock(&sound_mutex);
        kfree(preg);
        return minor;
    }
    //加入系统进行管理,打开文件时会使用.
    snd_minors[minor] = preg;
    //创建设备模型管理实体,其中设备号会创建相应的设备文件
    preg -> dev = device_create(sound_class,device,MKDEV(major,minor),
                            private_data,"%s",name);
```

```
        if(IS_ERR(preg->dev)){
            snd_minors[minor] = NULL;
            mutex_unlock(&sound_mutex);
            minor = PTR_ERR(preg->dev);
            kfree(preg);
            return minor;
        }

        mutex_unlock(&sound_mutex);
        return 0;
    }
```

所有的子类型设备都会在其 dev_register 中对 snd_register_device_for_dev 进行调用，从而完成相关的注册。还是以 control 子类型为例：

```
    static int snd_ctl_dev_register(struct snd_device * device)
    {
        struct snd_card * card = device->device_data;
        int err,cardnum;
        char name[16];

        if(snd_BUG_ON(!card))
                return -ENXIO;
        cardnum = card->number;
        if(snd_BUG_ON(cardnum < 0 || cardnum >= SNDRV_CARDS))
                return -ENXIO;
        //为设备文件提供名字
        sprintf(name,"controlC%i",cardnum);
        //这里会调用 snd_register_device_for_dev 来完成注册功能.
        if((err = snd_register_device(SNDRV_DEVICE_TYPE_CONTROL,card,-1,
                                        &snd_ctl_f_ops,card,name)) < 0)
                return err;
        return 0;
    }
```

可见其主要工作就是根据 card 号为设备文件指定名字，然后完成相关的注册工作。

接下来看看具体的 card 及子设备的初始化及注册流程。首先要创建 card 实体，相应的接口是 snd_card_create，详细分析如下：

```
    int snd_card_create(int idx,const char * xid,
                struct module * module,int extra_size,
                struct snd_card ** card_ret)
```

```
{
    struct snd_card * card;
    int err,idx2;

    if(snd_BUG_ON(!card_ret))
        return -EINVAL;
    * card_ret = NULL;

    if(extra_size < 0)
        extra_size = 0;
    //分配 card 实体的空间,如果驱动通过扩展 card 进行管理,便会指定 extra_size
    card = kzalloc(sizeof(* card) + extra_size,GFP_KERNEL);
    if(!card)
        return -ENOMEM;
    if(xid)
        strlcpy(card->id,xid,sizeof(card->id));
    err = 0;
    mutex_lock(&snd_card_mutex);
    //这里要选出合适的 card 号,如果 ID 为 -1 则先查看有无强制匹配关系,没有则分配
    //第一个没有占用的号
    if(idx < 0){
        for(idx2 = 0;idx2 < SNDRV_CARDS;idx2 ++)
            /* idx == -1 == 0xffff means: take any free slot */
            if(~snd_cards_lock & idx & 1 << idx2){
                if(module_slot_match(module,idx2)){
                    idx = idx2;
                    break;
                }
            }
    }
    if(idx < 0){
        for(idx2 = 0;idx2 < SNDRV_CARDS;idx2 ++)
            /* idx == -1 == 0xffff means: take any free slot */
            if(~snd_cards_lock & idx & 1 << idx2){
                if(!slots[idx2] || !*slots[idx2]){
                    idx = idx2;
                    break;
                }
            }
    }
    if(idx < 0)
        err = -ENODEV;
```

```
    else if(idx < snd_ecards_limit) {
        if(snd_cards_lock &(1 << idx))
            err = -EBUSY;                   /* invalid */
    } else if(idx >= SNDRV_CARDS)
        err = -ENODEV;
    if(err < 0) {
        mutex_unlock(&snd_card_mutex);
        goto __error;
    }
    //获得的 card 号标记占用
    snd_cards_lock |= 1 << idx;             /* lock it */
    if(idx >= snd_ecards_limit)
        snd_ecards_limit = idx + 1;         /* increase the limit */
    mutex_unlock(&snd_card_mutex);
    //赋值 card 号,并进行初始化
    card -> number = idx;
    card -> module = module;
    INIT_LIST_HEAD(&card -> devices);
    init_rwsem(&card -> controls_rwsem);
    rwlock_init(&card -> ctl_files_rwlock);
    INIT_LIST_HEAD(&card -> controls);
    INIT_LIST_HEAD(&card -> ctl_files);
    spin_lock_init(&card -> files_lock);
    INIT_LIST_HEAD(&card -> files_list);
    init_waitqueue_head(&card -> shutdown_sleep);
#ifdef CONFIG_PM
    mutex_init(&card -> power_lock);
    init_waitqueue_head(&card -> power_sleep);
#endif
    //初始化 card 的 control 子设备
    err = snd_ctl_create(card);
    if(err < 0) {
        snd_printk(KERN_ERR "unable to register control minors\n");
        goto __error;
    }
    //创建 card 的信息,通过 proc 文件系统输出
    err = snd_info_card_create(card);
    if(err < 0) {
        snd_printk(KERN_ERR "unable to create card info\n");
        goto __error_ctl;
    }
    //如果驱动有扩展的实体,则在 card 尾部开始,指针则要初始化
```

```
    if(extra_size > 0)
        card -> private_data = (char * )card + sizeof(struct snd_card);
     * card_ret = card;
    return 0;

      __error_ctl:
    snd_device_free_all(card,SNDRV_DEV_CMD_PRE);
     __error:
    kfree(card);
    return err;
}
```

通过分析可见，这里除了对 card 的空间及内容进行初始化外，最主要的就是对 control 子设备初始化并加入到 card 中进行管理。

分配了 card 之后，就要通过 snd_device_new 初始化并加入必要的子设备，由于 control 已经加入，必需的子设备就是 pcm（会根据需要创建 playback 和 capture 两种文件），通过 snd_pcm_new 初始化并加入到 card 中。最后通过 snd_card_register 来完成所有设备的注册工作。snd_card_register 细节如下：

```
int snd_card_register(struct snd_card * card)
{
    int err;
    if(snd_BUG_ON(!card))
        return -EINVAL;

    //这里创建设备模型相关实体,并不会为其创建设备文件
    if(!card -> card_dev){
        card -> card_dev = device_create(sound_class,card -> dev,
                                         MKDEV(0,0),card,
                                         "card%i",card -> number);
        if(IS_ERR(card -> card_dev))
            card -> card_dev = NULL;
    }

    //对所有的子设备进行注册操作
    if((err = snd_device_register_all(card)) < 0)
        return err;
    mutex_lock(&snd_card_mutex);
    //将 card 加入系统进行管理
    if(snd_cards[card -> number]){
        /* already registered */
        mutex_unlock(&snd_card_mutex);
```

```
        return 0;
    }
    snd_card_set_id_no_lock(card,card->id[0] == '\0' ? NULL : card->id);
    snd_cards[card->number] = card;
    mutex_unlock(&snd_card_mutex);
    init_info_for_card(card);
…
    //创建相关 sysfs 文件
    if(card->card_dev) {
        err = device_create_file(card->card_dev,&card_id_attrs);
        if(err < 0)
            return err;
        err = device_create_file(card->card_dev,&card_number_attrs);
        if(err < 0)
            return err;
    }

    return 0;
}
```

其中除了注册系统以及创建 sysfs 文件外，最主要的就是通过 snd_device_register_all 进行子设备的注册。snd_device_register_all 的操作很简单，就是遍历 card 的 devices 链表，并调用子设备的 dev_register 接口（最终调用 snd_register_device_for_dev 完成注册操作），将子设备的状态标记为 SNDRV_DEV_REGISTERED，从而完成整个的注册流程。至此应用层就可以使用相关的设备文件对设备进行操作了。

2. control 子设备

control 子设备主要负责 card 中所包含的各种功能的设置，这些设置并不是固定于子设备上的，而是属于功能型的，所以通过 card 直接进行管理，将一个控制项加入 card 的接口 snd_ctl_add，具体细节如下：

```
int snd_ctl_add(struct snd_card * card,struct snd_kcontrol * kcontrol)
{
    struct snd_ctl_elem_id id;
    unsigned int idx;
    int err = -EINVAL;

    if(!kcontrol)
        return err;
    if(snd_BUG_ON(!card || !kcontrol->info))
        goto error;
    id = kcontrol->id;
    down_write(&card->controls_rwsem);
```

```
    //检查 control 是否已经加入,主要检查类型,设备号,以及名字等信息
    if(snd_ctl_find_id(card,&id)){
        up_write(&card->controls_rwsem);
        snd_printd(KERN_ERR "control %i:%i:%i:%s:%i is already present\n",
                id.iface,
                id.device,
                id.subdevice,
                id.name,
                id.index);
        err = -EBUSY;
        goto error;
    }
    if(snd_ctl_find_hole(card,kcontrol->count) < 0){
        up_write(&card->controls_rwsem);
        err = -ENOMEM;
        goto error;
    }
    //添加到 card 的 control 列表中
    list_add_tail(&kcontrol->list,&card->controls);
    //相关计数增加
    card->controls_count += kcontrol->count;
    kcontrol->id.numid = card->last_numid + 1;
    card->last_numid += kcontrol->count;
    up_write(&card->controls_rwsem);
    //通知 control 的增加
    for(idx = 0;idx < kcontrol->count;idx++,id.index++,id.numid++)
        snd_ctl_notify(card,SNDRV_CTL_EVENT_MASK_ADD,&id);
    return 0;

error:
    snd_ctl_free_one(kcontrol);
    return err;
}
```

从以上代码可见，音频控制的主要结构就是 snd_kcontrol，详细内容如下：

```
struct snd_kcontrol{
    //用于 control 的连接
    struct list_head list;                  /* list of controls */
    //每个控制的基本信息,包括类型,所属设备,名字等,以及管理的编号
    struct snd_ctl_elem_id id;
    unsigned int count;                     /* count of same elements */
```

```
        //实例化的函数接口,用于特定的控制接口
        //获取信息接口
        snd_kcontrol_info_t * info;
        //读取当前具体的设置
        snd_kcontrol_get_t * get;
        //写设置的接口
        snd_kcontrol_put_t * put;
        //这里的 tlv 是 type - length - value 的缩写,通常包含控制中更完整的信息
        union{
            snd_kcontrol_tlv_rw_t * c;
            const unsigned int * p;
        } tlv;
        //通常是该控制特殊的值信息
        unsigned long private_value;
        //通常是该控制所属设备的相关信息
        void * private_data;
        void( * private_free)(struct snd_kcontrol * kcontrol);
        //一个控制对应多个实体时,这里是多个实体的内容,主要是与文件关联的属性
        //以及对应的访问控制
        struct snd_kcontrol_volatile vd[0];      /* volatile data */
    };
```

从分析中可知，每一个控制接口可以将多个相同功能统一进行管理，而其中的元素就使用 snd_ctl_elem_id 表示，count 表示整体的管理数目，针对每个控制实体都可能会对应不同用户任务来进行操作以及访问控制，这些都属于针对应用层变化的信息，通过统一的 snd_kcontrol_volatile 来进行管理。对应控制的内容、需要进行设备特殊的操作，这些都是通过 info、get、put 以及 private_value 和 private_data 来共同实现的。所以这里包括了针对上层应用层的管理部分，以及针对下层驱动的操作属性。

应用层更关注的是控制元素的信息以及相关值的属性，基本都是通过 snd_ctl_elem_xxx 来获得的，这些结构定义都在 asound. h 中。

所有的控制都是底层驱动提供的，所以控制信息应该由底层驱动进行定义，而由于 snd_kcontrol 中包含了针对上层的管理信息，相应的并不适用于底层驱动直接定义控制信息。针对这种情况，ALSA 框架提供了 snd_kcontrol_new 来进行定义，详细内容如下：

```
    struct snd_kcontrol_new{
        //控制元素的类型
        snd_ctl_elem_iface_t iface;              /* interface identifier */
        //设备号
        unsigned int device;                     /* device/client number */
        //子设备
        unsigned int subdevice;                  /* subdevice(substream)number */
```

```
    unsigned char * name;                /* ASCII name of item */
    unsigned int index;                  /* index of item */
    //访问控制
    unsigned int access;                 /* access rights */
    unsigned int count;                  /* count of same elements */
    //设备特殊的操作接口
    snd_kcontrol_info_t * info;
    snd_kcontrol_get_t * get;
    snd_kcontrol_put_t * put;
    union{
        snd_kcontrol_tlv_rw_t * c;
        const unsigned int * p;
    } tlv;
    //控制特殊的值信息,可以赋予指针值,设备操作接口会进行相应转换
    unsigned long private_value;
};
```

从其结构体内容可见，其中主要是控制与设备信息的设置，符合底层的数据特点。这样就需要接口能将底层使用的实体转换为上层使用的实体，这个接口就是 snd_ctl_new1，其定义如下：

```
struct snd_kcontrol * snd_ctl_new1(const struct snd_kcontrol_new * kcontrolnew,
                                   void * private_data);
```

实际应用中通常将 snd_ctl_add 与 snd_ctl_new1 结合，就可以完成每个控制元素的注册。驱动部分的控制元素只要完成注册就可以了，而在上层中针对应用层的处理则由 ALSA 提供的 control 子设备的操作接口来完成，主要将每个 kcontrol 的元素看做 element，而通过 snd_ctl_elem_xxx 的接口进行遍历、读、写等操作（这些操作都是 ALSA 应用层的 ioctl 命令）。这样就完成了整个的控制流程，可见在注册之后控制流的调用层次是很少的，主要因为控制操作本来就是很直接的工作，设计上也体现了这种直接性的特点。

利用 kcontrol，可以完成对音频系统中的 mixer、mux、音量控制、音效控制以及各种开关量的控制，这些操作都是通过对各种不同类型 kcontrol（不同类型的控制元素）的操作来完成的，通过这些控制可以使得音频设备按照应用的设计进行工作，完成各种音频需求。

以上是 control 子设备中包含的各模块功能的 kcontrl 控制部分，还有一部分就是对 card 其他类型子设备的查询等控制也是要通过 control 子设备来统一完成，这样可以有统一的接口来遍历整个音频设备 card 的情况，相应的可以从 snd_ctl_ioctl 来了解它是如何工作的。

```
static long snd_ctl_ioctl(struct file * file,unsigned int cmd,unsigned long arg)
{
```

```
struct snd_ctl_file * ctl;
struct snd_card * card;
struct snd_kctl_ioctl * p;
void __user * argp = (void __user * )arg;
int __user * ip = argp;
int err;

ctl = file -> private_data;
card = ctl -> card;
if(snd_BUG_ON(!card))
    return -ENXIO;
//这里是 kcontrol 的操作接口
switch(cmd) {
case SNDRV_CTL_IOCTL_PVERSION:
    return put_user(SNDRV_CTL_VERSION,ip) ? -EFAULT : 0;
case SNDRV_CTL_IOCTL_CARD_INFO:
    return snd_ctl_card_info(card,ctl,cmd,argp);
case SNDRV_CTL_IOCTL_ELEM_LIST:
    return snd_ctl_elem_list(card,argp);
case SNDRV_CTL_IOCTL_ELEM_INFO:
    return snd_ctl_elem_info_user(ctl,argp);
case SNDRV_CTL_IOCTL_ELEM_READ:
    return snd_ctl_elem_read_user(card,argp);
case SNDRV_CTL_IOCTL_ELEM_WRITE:
    return snd_ctl_elem_write_user(ctl,argp);
case SNDRV_CTL_IOCTL_ELEM_LOCK:
    return snd_ctl_elem_lock(ctl,argp);
case SNDRV_CTL_IOCTL_ELEM_UNLOCK:
    return snd_ctl_elem_unlock(ctl,argp);
case SNDRV_CTL_IOCTL_ELEM_ADD:
    return snd_ctl_elem_add_user(ctl,argp,0);
case SNDRV_CTL_IOCTL_ELEM_REPLACE:
    return snd_ctl_elem_add_user(ctl,argp,1);
case SNDRV_CTL_IOCTL_ELEM_REMOVE:
    return snd_ctl_elem_remove(ctl,argp);
case SNDRV_CTL_IOCTL_SUBSCRIBE_EVENTS:
    return snd_ctl_subscribe_events(ctl,ip);
case SNDRV_CTL_IOCTL_TLV_READ:
    return snd_ctl_tlv_ioctl(ctl,argp,0);
case SNDRV_CTL_IOCTL_TLV_WRITE:
    return snd_ctl_tlv_ioctl(ctl,argp,1);
case SNDRV_CTL_IOCTL_TLV_COMMAND:
```

```
            return snd_ctl_tlv_ioctl(ctl,argp, -1);
        case SNDRV_CTL_IOCTL_POWER:
            return -ENOPROTOOPT;
        case SNDRV_CTL_IOCTL_POWER_STATE:
#ifdef CONFIG_PM
            return put_user(card->power_state,ip) ? -EFAULT : 0;
#else
            return put_user(SNDRV_CTL_POWER_D0,ip) ? -EFAULT : 0;
#endif
    }
    down_read(&snd_ioctl_rwsem);
    //其他类型子设备的 ioctl 接口,用来获得其他子设备的情况
    list_for_each_entry(p,&snd_control_ioctls,list) {
        err = p->fioctl(card,ctl,cmd,arg);
        if(err != -ENOIOCTLCMD) {
            up_read(&snd_ioctl_rwsem);
            return err;
        }
    }
    up_read(&snd_ioctl_rwsem);
    snd_printdd("unknown ioctl = 0x%x\n",cmd);
    return -ENOTTY;
}
```

从代码中可以了解 kcontrol 的操作方法，另外还提供了获得其他子设备情况的接口，这需要其他类型子设备如 pcm 向系统注册操作接口，通过 snd_ctl_register_ioctl 加入链表 snd_control_ioctls 进行注册。这样的设计可以保证应用通过 control 子设备来查询音频设备的整体状况并进行相关的操作，这样可以有统一的操作界面，适合应用的需求。

3. pcm 子设备

前面介绍了音频设备的控制流程，接下来看看音频设备的 pcm 数据流程。对 pcm 数据流，ALSA 设计了统一的管理方式，其中为下层驱动提供的添加 pcm 数据通道的接口函数 snd_pcm_new，详细内容以及分析如下：

```
int snd_pcm_new(struct snd_card *card,const char *id,int device,
        int playback_count,int capture_count,
            struct snd_pcm **rpcm)
{
    struct snd_pcm *pcm;
    int err;
    //设备管理接口
    static struct snd_device_ops ops = {
        .dev_free = snd_pcm_dev_free,
```

```
        . dev_register =        snd_pcm_dev_register,
        . dev_disconnect = snd_pcm_dev_disconnect,
    };

    if(snd_BUG_ON(!card))
        return -ENXIO;
    if(rpcm)
        * rpcm = NULL;
    //分配管理实体 snd_pcm
    pcm = kzalloc(sizeof( * pcm),GFP_KERNEL);
    if(pcm == NULL){
        snd_printk(KERN_ERR "Cannot allocate PCM\n");
        return -ENOMEM;
    }
    //与 card 关联
    pcm -> card = card;
    //记录子类型设备的 index 号
    pcm -> device = device;
    //记录设备名
    if(id)
        strlcpy(pcm -> id,id,sizeof(pcm -> id));
    //添加管理的子 playback 流,可以是多个
    if((err = snd_pcm_new_stream(pcm,SNDRV_PCM_STREAM_PLAYBACK,playback_count)) <
0){
        snd_pcm_free(pcm);
        return err;
    }
    //添加管理的子 capture 流,可以是多个
    if((err = snd_pcm_new_stream(pcm,SNDRV_PCM_STREAM_CAPTURE,capture_count)) <0){
        snd_pcm_free(pcm);
        return err;
    }
    mutex_init(&pcm -> open_mutex);
    //应用层操作的打开控制,主要是避免多次打开,因为音频数据流比较敏感
    //数据一致性
    init_waitqueue_head(&pcm -> open_wait);
    //加入 card 进行管理,后续的 card register 会调用 snd_pcm_dev_register 注册
    if((err = snd_device_new(card,SNDRV_DEV_PCM,pcm,&ops)) <0){
        snd_pcm_free(pcm);
        return err;
    }
    if(rpcm)
```

```
        *rpcm = pcm;
    return 0;
}
```

以上是驱动加入 pcm 流的过程，而具体的注册是在 card register 中调用子设备的 snd_pcm_dev_register 来完成的。具体内容如下：

```
static int snd_pcm_dev_register(struct snd_device *device)
{
    int cidx,err;
    struct snd_pcm_substream *substream;
    struct snd_pcm_notify *notify;
    char str[16];
    struct snd_pcm *pcm;
    struct device *dev;

    if(snd_BUG_ON(!device || !device->device_data))
        return -ENXIO;
    //通过 snd_device 获得 pcm 管理实体
    pcm = device->device_data;
    mutex_lock(&register_mutex);
    //加入到系统中允许通过 control 子设备查询 pcm 信息
    err = snd_pcm_add(pcm);
    if(err){
        mutex_unlock(&register_mutex);
        return err;
    }
    //分别对 playback 和 capture 进行不同的操作
    for(cidx =0;cidx <2;cidx ++){
        int devtype = -1;
        if(pcm->streams[cidx].substream == NULL)
            continue;
        //为 playback 和 capture 设备产生合适的设备名
        switch(cidx){
        case SNDRV_PCM_STREAM_PLAYBACK:
            sprintf(str,"pcmC%iD%ip",pcm->card->number,pcm->device);
            devtype = SNDRV_DEVICE_TYPE_PCM_PLAYBACK;
            break;
        case SNDRV_PCM_STREAM_CAPTURE:
            sprintf(str,"pcmC%iD%ic",pcm->card->number,pcm->device);
            devtype = SNDRV_DEVICE_TYPE_PCM_CAPTURE;
            break;
```

```
        }
        //这里通常是该 pcm 的物理设备即父设备,一般都是空,所以父设备
        //会是 card device
        dev = pcm -> dev;
        if( ! dev)
            dev = snd_card_get_device_link( pcm -> card) ;
        / * register pcm * /
        //注册 pcm 到系统中并会根据信息产生设备文件,设备文件操作接口
        //会根据 playback 与 capture 有不同 .
        err = snd_register_device_for_dev( devtype, pcm -> card,
                                 pcm -> device,
                                 &snd_pcm_f_ops[ cidx],
                                 pcm, str, dev) ;
        if( err < 0) {
            list_del( &pcm -> list) ;
            mutex_unlock( &register_mutex) ;
            return err;
        }
        //增加 sysfs 文件
        snd_add_device_sysfs_file( devtype, pcm -> card, pcm -> device,
                        &pcm_attrs) ;
            //这里为所有的数据流创建向应用层提供信息的 timer,应用层可以通过
            //对系统的 timer 文件来对音频流的时间进行监控 . 一般不使用该功能 .
            for( substream = pcm -> streams[ cidx]. substream; substream; substream = substream ->
                             next)
            snd_pcm_timer_init( substream) ;
    }

    //通知 pcm 已经注册事件
    list_for_each_entry( notify, &snd_pcm_notify_list, list)
        notify -> n_register( pcm) ;

    mutex_unlock( &register_mutex) ;
    return 0;
}
```

从添加和注册的细节可见，ALSA 框架中对 pcm 进行了统一的管理，由 snd_pcm 来提供，而其中会根据 playback 和 capture 来区分为不同类型的 snd_pcm_str，在不同类型 snd_pcm_str 中则可以有多个 substream 由 snd_pcm_substream 进行管理。这些管理实体的关系如图 6-15 所示。

所有这些管理实体 snd_pcm 和 snd_pcm_str 更偏向于组织，而 snd_pcm_substream 是直接对应到实际的音频流，可以说是音频设备数据流的核心。下面看一下 snd_pcm_

substream 的详细内容。

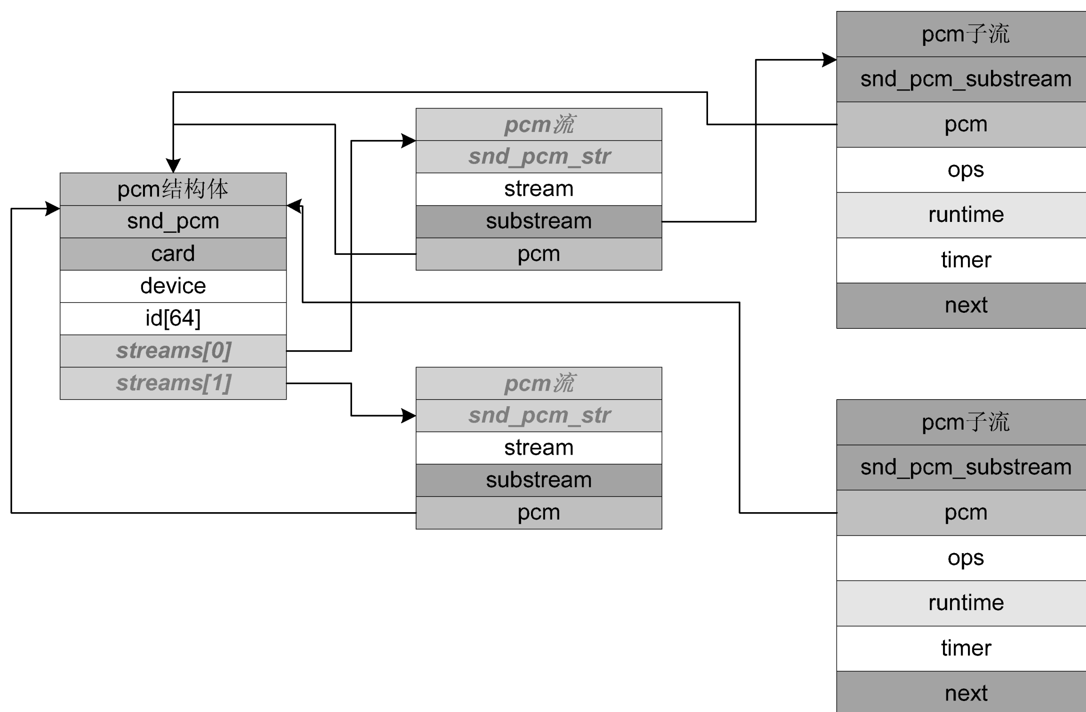


图 6-15　pcm 中管理实体关系

```
struct snd_pcm_substream{
    //指向整体 pcm 管理实体
    struct snd_pcm * pcm;
    //指向特定方向 stream 的管理实体(playback 或者 capture)
    struct snd_pcm_str * pstr;
    //驱动相关的实体,主要是总线框架使用如 USB
    void * private_data;                    /* copied from pcm -> private_data */
    //同方向 stream 的 index 号
    int number;
    char name[32];                          /* substream name */
    //表示 stream 的方向 playback 或者 capture
    int stream;                             /* stream(direction) */
    struct pm_qos_request_list latency_pm_qos_req;   /* pm_qos request */
    size_t buffer_bytes_max;                /* limit ring buffer size */
    //分配的 DMA 空间的信息,一般音频流是通过 DMA 传输的
    struct snd_dma_buffer dma_buffer;
    //如果使用 reserve 空间才设置,通常没有为音频保留空间,所以不设置该值
    unsigned int dma_buf_id;
    //设置的 DMA 分配的最大空间
    size_t dma_max;
```

```
        /* --hardware operations-- */
        //硬件相关的操作
        struct snd_pcm_ops *ops;
        /* --runtime information-- */
        //运行时 pcm 流的状态,包括当前位置、时间戳等信息
        struct snd_pcm_runtime *runtime;
            /* --timer section-- */
        //由于音频流会按照一定时间间隔传送,所以关联到向应用层通知时间事件
        struct snd_timer *timer;            /* timer */
        //设置表示启动 timer 功能
        unsigned timer_running: 1;          /* time is running */
        /* --next substream-- */
        //用于连接 substream
        struct snd_pcm_substream *next;
        /* --linked substreams-- */
        //alsa 可以将多个 stream 关联起来,然后统一的进行操作,以下就是实现
        //该部分功能的相关属性
        struct list_head link_list;         /* linked list member */
        struct snd_pcm_group self_group;    /* fake group for non linked substream(with substream
                                               lock inside) */
        struct snd_pcm_group *group;        /* pointer to current group */
        /* --assigned files-- */
        void *file;
        int ref_count;
        //mmap 打开的数目
        atomic_t mmap_count;
        //记录文件的操作属性,以便在该层进行相应的操作
        unsigned int f_flags;
        void( *pcm_release)(struct snd_pcm_substream *);
        struct pid *pid;
...
        /* misc flags */
        unsigned int hw_opened: 1;
};
```

重要的管理实体是snd_pcm_runtime，其中包含了运行时的各种状态和属性，与音频流息息相关。具体内容如下：

```
struct snd_pcm_runtime{
        /* --Status-- */
        //记录各种状态信息
        //属于哪个 substream 触发
```

```
struct snd_pcm_substream * trigger_master;
//各种触发事件的时间
struct timespec trigger_tstamp;              /* trigger timestamp */
int overrange;
snd_pcm_uframes_t avail_max;
snd_pcm_uframes_t hw_ptr_base;               /* Position at buffer restart */
snd_pcm_uframes_t hw_ptr_interrupt;          /* Position at interrupt time */
unsigned long hw_ptr_jiffies;                /* Time when hw_ptr is updated */
unsigned long hw_ptr_buffer_jiffies;         /* buffer time in jiffies */
snd_pcm_sframes_t delay;                     /* extra delay;typically FIFO size */

/* -- HW params -- */
//硬件的各种参数
snd_pcm_access_t access;                     /* access mode */
//采样数据 word 的格式
snd_pcm_format_t format;                     /* SNDRV_PCM_FORMAT_* */
snd_pcm_subformat_t subformat;               /* subformat */
//采样率
unsigned int rate;                           /* rate in Hz */
//channel 数目
unsigned int channels;                       /* channels */
snd_pcm_uframes_t period_size;               /* period size */
unsigned int periods;                        /* periods */
snd_pcm_uframes_t buffer_size;               /* buffer size */
snd_pcm_uframes_t min_align;                 /* Min alignment for the format */
size_t byte_align;
unsigned int frame_bits;
unsigned int sample_bits;
unsigned int info;
unsigned int rate_num;
unsigned int rate_den;

/* -- SW params -- */
int tstamp_mode;                             /* mmap timestamp is updated */
unsigned int period_step;
snd_pcm_uframes_t start_threshold;
snd_pcm_uframes_t stop_threshold;
snd_pcm_uframes_t silence_threshold;         /* Silence filling happens when
                         noise is nearest than this */
snd_pcm_uframes_t silence_size;              /* Silence filling size */
snd_pcm_uframes_t boundary;                  /* pointers wrap point */
```

```
snd_pcm_uframes_t silence_start;               /* starting pointer to silence area */
snd_pcm_uframes_t silence_filled;              /* size filled with silence */

union snd_pcm_sync_id sync;                    /* hardware synchronization ID */

/* -- mmap -- */
//以下信息可以通过 mmap 命令 map 到用户空间供用户获得状态信息并使用
//运行时内存空间的操作状态包括地址位置、时间戳等
struct snd_pcm_mmap_status *status;
//应用层操作的偏移信息等
struct snd_pcm_mmap_control *control;

/* -- locking/scheduling -- */
snd_pcm_uframes_t twake;                       /* do transfer(!poll) wakeup if non-zero */
wait_queue_head_t sleep;                       /* poll sleep */
wait_queue_head_t tsleep;                      /* transfer sleep */
struct fasync_struct *fasync;

/* -- private section -- */
void *private_data;
void(*private_free)(struct snd_pcm_runtime *runtime);

/* -- hardware description -- */
//硬件信息描述
struct snd_pcm_hardware hw;
struct snd_pcm_hw_constraints hw_constraints;

/* -- interrupt callbacks -- */
void(*transfer_ack_begin)(struct snd_pcm_substream *substream);
void(*transfer_ack_end)(struct snd_pcm_substream *substream);

/* -- timer -- */
unsigned int timer_resolution;                 /* timer resolution */
int tstamp_type;                               /* timestamp type */

/* -- DMA -- */
//DMA 使用的属性,包括内存空间描述等信息
unsigned char *dma_area;                       /* DMA area */
dma_addr_t dma_addr;                           /* physical bus address(not accessible from main
                                                  CPU) */
size_t dma_bytes;                              /* size of DMA area */
```

```
        struct snd_dma_buffer * dma_buffer_p;      /* allocated buffer */
...
};
```

可见其很多属性是从其他管理实体中复制过来的，比如 dma 空间的属性，以及其中一部分硬件的属性等，这是为了操作的方便；runtime 的状态更新涉及上层应用和底层驱动，主要是运行时的状态、音频流状态的更新，系统提供了统一的接口进行相应的更新操作，如 snd_pcm_update_hw_ptr 和 snd_pcm_period_elapsed（驱动进行更新的接口）。下面来看看 snd_pcm_period_elapsed 的具体内容：

```
void snd_pcm_period_elapsed(struct snd_pcm_substream * substream)
{
        struct snd_pcm_runtime * runtime;
        unsigned long flags;

        if(PCM_RUNTIME_CHECK(substream))
                return;
        runtime = substream->runtime;

        if(runtime->transfer_ack_begin)
                runtime->transfer_ack_begin(substream);

        snd_pcm_stream_lock_irqsave(substream,flags);
        //检查是否 pcm 处于传输状态,如是,则根据 runtime 的信息更新相关状态,主要是
        //内存地址和时间戳等信息
        if(!snd_pcm_running(substream) ||
                snd_pcm_update_hw_ptr0(substream,1) <0)
                goto _end;

        //更新 timer 相关信息
        if(substream->timer_running)
                snd_timer_interrupt(substream->timer,1);
_end:
        snd_pcm_stream_unlock_irqrestore(substream,flags);
        if(runtime->transfer_ack_end)
                runtime->transfer_ack_end(substream);
        //异步通知应用层,一般 mmap 方式需要
        kill_fasync(&runtime->fasync,SIGIO,POLL_IN);
}
```

这里有一个重要的概念就是周期（period），这是因为一次 DMA 操作唤醒一次中断太浪费系统资源，所以通常定义一个数据块大小表示一个周期可以传送数据的大小，这样的设计

考虑整体的系统性能。

对驱动来说，重要接口是设置 snd_pcm_runtime 中的 snd_pcm_ops。snd_pcm_ops 的具体内容如下：

```
struct snd_pcm_ops{
    int( * open)(struct snd_pcm_substream * substream);
    int( * close)(struct snd_pcm_substream * substream);
    //ioctl 的接口
    int( * ioctl)(struct snd_pcm_substream * substream,
            unsigned int cmd,void * arg);
    //在传输之前进行硬件参数设置的接口
    int( * hw_params)(struct snd_pcm_substream * substream,
            struct snd_pcm_hw_params * params);
    //硬件释放的接口
    int( * hw_free)(struct snd_pcm_substream * substream);
    //传输之前的准备操作的接口
    int( * prepare)(struct snd_pcm_substream * substream);
    //各种状态转换的接口
    int( * trigger)(struct snd_pcm_substream * substream,int cmd);
    //获得当前硬件操作 DMA 空间指针状态,返回数据的当前读取位置(capture)
    //或数据的当前写入位置(playback)
    snd_pcm_uframes_t( * pointer)(struct snd_pcm_substream * substream);
    //硬件进行数据拷贝的接口
    int( * copy)(struct snd_pcm_substream * substream,int channel,
            snd_pcm_uframes_t pos,
            void __user * buf,snd_pcm_uframes_t count);
    int( * silence)(struct snd_pcm_substream * substream,int channel,
            snd_pcm_uframes_t pos,snd_pcm_uframes_t count);
    //内存空间缺页异常的操作接口
    struct page * ( * page)(struct snd_pcm_substream * substream,
            unsigned long offset);
    //进行 mmap 操作接口
    int( * mmap)(struct snd_pcm_substream * substream,struct vm_area_struct * vma);
    //应用读写需要硬件进行同步的接口
    int( * ack)(struct snd_pcm_substream * substream);
};
```

驱动主要就是实现这些接口，并通过 snd_pcm_set_ops 进行设置。

而应用层的各种操作会使得 PCM 流在不同的状态中转换，具体的状态如下：

```
#define    SNDRV_PCM_STATE_OPEN        ((__force snd_pcm_state_t)0)
#define    SNDRV_PCM_STATE_SETUP       ((__force snd_pcm_state_t)1)
#define    SNDRV_PCM_STATE_PREPARED    ((__force snd_pcm_state_t)2)
```

```
#define    SNDRV_PCM_STATE_RUNNING         ((__force snd_pcm_state_t)3)
#define    SNDRV_PCM_STATE_XRUN            ((__force snd_pcm_state_t)4)
#define    SNDRV_PCM_STATE_DRAINING        ((__force snd_pcm_state_t)5)
#define    SNDRV_PCM_STATE_PAUSED          ((__force snd_pcm_state_t)6)
#define    SNDRV_PCM_STATE_SUSPENDED       ((__force snd_pcm_state_t)7)
#define    SNDRV_PCM_STATE_DISCONNECTED    ((__force snd_pcm_state_t)8)
```

ALSA 的框架会针对不同的操作调用合适的驱动 snd_pcm_ops 接口函数，从而完成合适的操作。

对整体的 pcm 子设备的理解，同样要分为控制部分和数据部分，控制部分就是各种命令使得 pcm 流在不同状态间进行转换。基本的控制流程是用户通过 open 打开子码流并获得该子码流的能力参数，从而确定出一套可用的参数；调用 hw_params 设置这些参数；调用 prepare 进行最后的准备工作，比如清除 fifo；调用 trigger 启动或停止工作；通过 pointer 了解工作进展；工作停止后，调用 hw_free 释放资源，使用 close 关闭子码流。而数据部分是通过 runtime 进行状态的维护和同步，由应用层的 read、write 或者 mmap 方式和驱动（DMA 的方式）共同操作 ring buffer 的空间，从而完成整个的数据操作。

对数据部分，应用层 read 和 write 音频流数据需要考虑数据的组织形式，通过不同的 ioctl 命令来完成，当多个 channel 的音频数据放在同一块连续存储空间时是属于 interleave 方式存放，需要进行 readi 和 writei 操作；否则就是不同的 channel 数据放入分离的存储空间中，需要通过 readn 和 writen 进行操作。应用层需要根据数据的具体属性进行相关的操作。这样框架层就完成了数据的操作，其余的数据操作属于驱动的实现部分。

4. ASoC（ALSA SoC）子框架

以上的框架主要适合于 PC 上的音频设备，这些音频设备的特定是数字部分和模拟部分集成在一起，所以通过统一的控制就可以完成。而随着嵌入式的发展，SoC 处理器大量地出现，这些 SoC 的特点是在片上包含音频数据流的传输通道，而具体的音频处理以及模拟部分是在 SoC 外部的音频 codec 进行处理的，这就使通道和处理是分离的，这样设计的特点是使得数字部分和模拟部分隔离，降低芯片开发的难度，也可以集成不同的 codec 来完成不同的功能。在软件上这就带来了问题，之前的架构对 SoC 来说并不适用。如果使用原有的音频框架，要完成 SoC 音频数据接口的驱动还需要包括音频 codec 的操作，这样才是完整的音频设备，这样就不能在不同的 SoC 之间重用音频 codec 的驱动，从而提高了系统的复杂程度。为了解决这些问题，ALSA 提供了 ASoC 子架构，如图 6-16 所示。

从图 6-16 可见，ALSA SoC 将不同的设备主要分为 machine、codec 和 platform 几个部分。codec 主要负责 audio codec 部分；platform 主要负责音频流的传输控制；machine 则是描述相关的连接，使得 ALSA SoC 框架可以将这些分离的组件进行正确的连接。另外 audio codec 和 SoC 都包含数据通道的控制，在 ALSA SoC 框架中称为 DAI（digital audio interfaces）这部分的属性和控制单独进行管理，在 audio codec 和 platform 双方都有相应的管理实体和操作。这样整体上系统就可以分离成各种不同的管理实体，各种实体可以分别进行驱动的开发，最终通过 machine（相当于板级的连接管理）将这些实体关联成系统，从而使得整个系统正常工作。

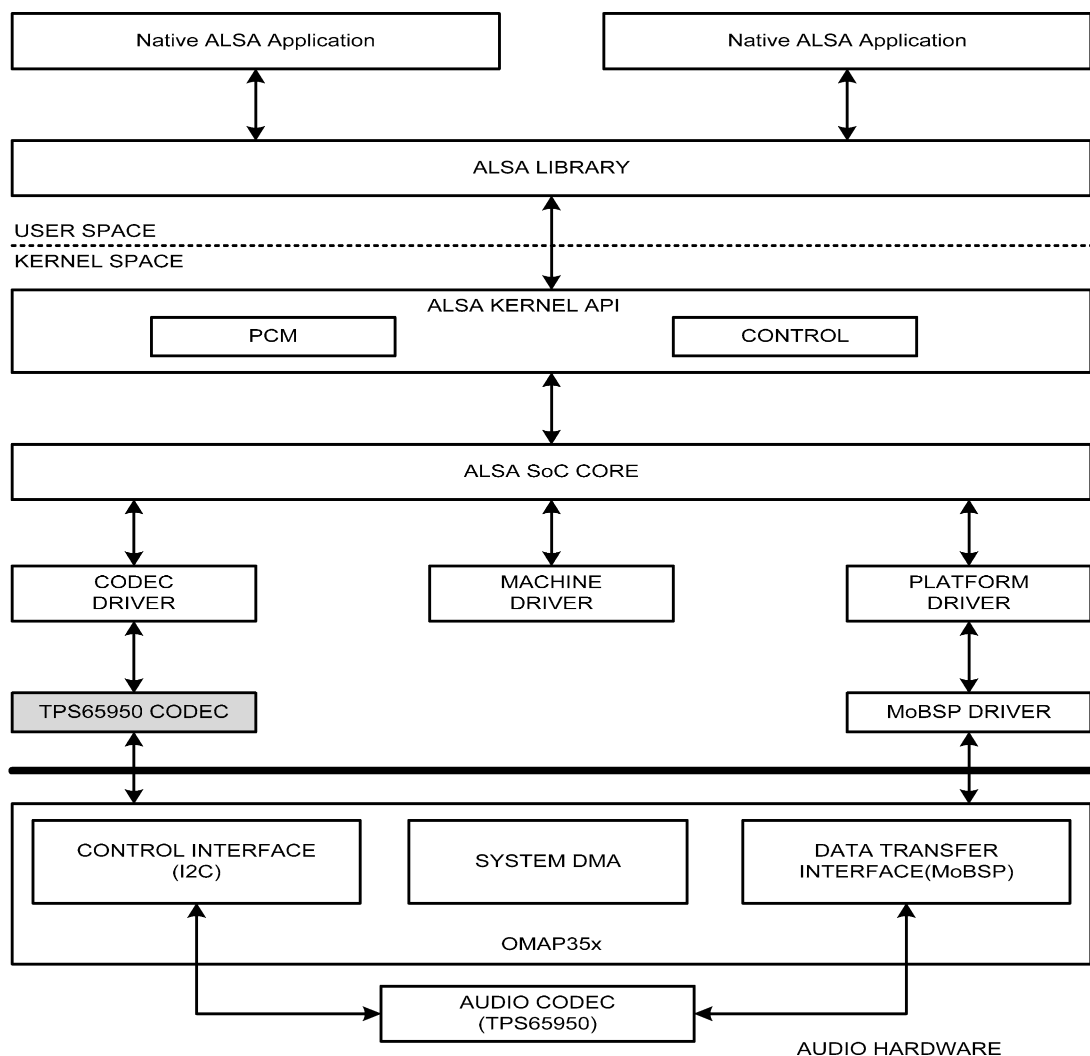


图 6-16　alsa soc 框架

ALSA SoC 框架针对以上的实体分别提供了不同的管理结构，具体如下：

- snd_soc_dai：用于描述 SoC 处理器端或 codec 端接口的硬件能力。在运行过程中，ASoC 框架会综合两端的硬件能力，生成两端都可接收硬件能力参数（snd_pcm_hardware）。
- snd_soc_platform：用于管理 SoC 处理器端数据流的操作。
- snd_soc_dai_link：用于描述 SoC 处理器端和 codec 端两端的实际硬件连接。即 SoC 处理器端哪个硬件通道与哪个 codec 进行连接。
- snd_soc_card：用于描述所有的硬件连接，最终会建立一个 card，而每个连接会建立一个 pcm 通道。
- snd_soc_codec：用于描述 audio codec 属性，以及相关操作。

而以上所有这些硬件连接的各个模块都通过 snd_soc_pcm_runtime 进行关联，其内容如下：

```
struct snd_soc_pcm_runtime {
    struct device dev;
    struct snd_soc_card * card;
    struct snd_soc_dai_link * dai_link;

    unsigned int complete:1;
    unsigned int dev_registered:1;

    /* Symmetry data - only valid if symmetry is being enforced */
    unsigned int rate;
    long pmdown_time;

    /* runtime devices */
    struct snd_pcm * pcm;
    struct snd_soc_codec * codec;
    struct snd_soc_platform * platform;
    struct snd_soc_dai * codec_dai;
    struct snd_soc_dai * cpu_dai;

    struct delayed_work delayed_work;
};
```

从结构的实际属性可见，其中将 ALSA SoC 下层具体的功能模块与上层 ALSA 的 pcm 流进行了关联，这样就建立了上层与下层的连接，保证操作的畅通。

ALSA SoC 框架则提供将底层各个模块关联的操作，通过 soc_bind_dai_link 来进行，详细分析如下：

```
static int soc_bind_dai_link(struct snd_soc_card * card,int num)
{
    struct snd_soc_dai_link * dai_link = &card->dai_link[num];
    struct snd_soc_pcm_runtime * rtd = &card->rtd[num];
    struct snd_soc_codec * codec;
    struct snd_soc_platform * platform;
    struct snd_soc_dai * codec_dai, * cpu_dai;
    //已经绑定则直接返回
    if(rtd->complete)
        return 1;
    dev_dbg(card->dev,"binding %s at idx %d\n",dai_link->name,num);

    /* do we already have the CPU DAI for this link ? */
    //绑定处理器侧的 DAI
    if(rtd->cpu_dai){
```

```
            goto find_codec;
        }
        /* no, then find CPU DAI from registered DAIs */
        list_for_each_entry(cpu_dai, &dai_list, list) {
            if(!strcmp(cpu_dai->name, dai_link->cpu_dai_name)) {

                if(!try_module_get(cpu_dai->dev->driver->owner))
                    return -ENODEV;

                rtd->cpu_dai = cpu_dai;
                goto find_codec;
            }
        }
        dev_dbg(card->dev, "CPU DAI %s not registered\n",
                dai_link->cpu_dai_name);

find_codec:
        /* do we already have the CODEC for this link? */
        //绑定 codec
        if(rtd->codec) {
            goto find_platform;
        }

        /* no, then find CODEC from registered CODECs */
        //查询每个注册的 codec
        list_for_each_entry(codec, &codec_list, list) {
            if(!strcmp(codec->name, dai_link->codec_name)) {
                //绑定 codec
                rtd->codec = codec;

                if(!try_module_get(codec->dev->driver->owner))
                    return -ENODEV;

            /* CODEC found, so find CODEC DAI from registered DAIs from this CODEC */
                //查找并绑定 codec 侧的 DAI
                list_for_each_entry(codec_dai, &dai_list, list) {
                    if(codec->dev == codec_dai->dev &&
                            !strcmp(codec_dai->name, dai_link->codec_dai_name)) {
                        rtd->codec_dai = codec_dai;
                        goto find_platform;
                    }
                }
```

```
                dev_dbg(card -> dev,"CODEC DAI %s not registered\n",
                        dai_link -> codec_dai_name);

                goto find_platform;
            }
        }
        dev_dbg(card -> dev,"CODEC %s not registered\n",
                dai_link -> codec_name);

find_platform:
        //绑定处理器侧的数据操作管理实体 platform
        if(rtd -> platform) {
            goto out;
        }
        /* no,then find CPU DAI from registered DAIs */
        list_for_each_entry(platform,&platform_list,list) {
            if(!strcmp(platform -> name,dai_link -> platform_name)) {

                if(!try_module_get(platform -> dev -> driver -> owner))
                    return -ENODEV;

                rtd -> platform = platform;
                goto out;
            }
        }

        dev_dbg(card -> dev,"platform %s not registered\n",
                dai_link -> platform_name);
        return 0;

out:
        /* mark rtd as complete if we found all 4 of our client devices */
        //所有需要的模块都已经绑定则表示绑定完成,设置标识.
        if(rtd -> codec && rtd -> codec_dai && rtd -> platform && rtd -> cpu_dai) {
            rtd -> complete = 1;
            card -> num_rtd++;
        }
        return 1;
}
```

从代码中可见，只有 rtd -> codec、rtd -> codec_dai、rtd -> platform 和 rtd -> cpu_dai 都绑定才是真正的绑定成功，从而进行后续操作。对于每种组件 ALSA SoC 框架都提供注册接

口，接口如下：

```
int snd_soc_register_dai(struct device * dev,struct snd_soc_dai_driver * dai_drv);
int snd_soc_register_codec(struct device * dev,struct snd_soc_codec_driver * codec_drv,
        struct snd_soc_dai_driver * dai_drv,int num_dai);
int snd_soc_register_platform(struct device * dev,
        struct snd_soc_platform_driver * platform_drv);
```

各个组件会在内部通过以上接口向 ALSA SoC 注册相应的组件。为了适应各种不同的组件注册情况，系统提供了整体的 card 级别的初始化操作，并会在任何组件注册时被调用，保证系统能正确地发现不同的组件并关联。这个逻辑最终由 snd_soc_instantiate_card 来完成，详细内容如下：

```
static void snd_soc_instantiate_card(struct snd_soc_card * card)
{
    struct platform_device * pdev = to_platform_device(card->dev);
    int ret,i;

    mutex_lock(&card->mutex);

    //已经完成实例化
    if(card->instantiated){
        mutex_unlock(&card->mutex);
        return;
    }

    /* bind DAIs */
    //进行所有物理连接的各个模块绑定
    for(i=0;i<card->num_links;i++)
        soc_bind_dai_link(card,i);

    /* bind completed ? */
    //所有物理连接都完成绑定,才能继续
    if(card->num_rtd != card->num_links){
        mutex_unlock(&card->mutex);
        return;
    }

    /* card bind complete so register a sound card */
    //创建 ALSA 层的 card 管理实体
    ret = snd_card_create(SNDRV_DEFAULT_IDX1,SNDRV_DEFAULT_STR1,
            card->owner,0,&card->snd_card);
    if(ret<0){
```

```
            printk(KERN_ERR "asoc: can't create sound card for card %s\n",
                card->name);
            mutex_unlock(&card->mutex);
            return;
        }
        //设备模型的层次关系
        card->snd_card->dev = card->dev;

#ifdef CONFIG_PM
        /* deferred resume work */
        INIT_WORK(&card->deferred_resume_work,soc_resume_deferred);
#endif

        /* initialise the sound card only once */
        if(card->probe){
            ret = card->probe(pdev);
            if(ret < 0)
                goto card_probe_error;
        }

        for(i = 0;i < card->num_links;i++){
            //检查物理连接的各个组件并进行探测操作,进行必要的初始化添加 control
            //最终还会通过 soc_new_pcm 创建 pcm 实体
            ret = soc_probe_dai_link(card,i);
            if(ret < 0){
                pr_err("asoc: failed to instantiate card %s: %d\n",
                    card->name,ret);
                goto probe_dai_err;
            }
        }

        snprintf(card->snd_card->shortname,sizeof(card->snd_card->shortname),
             "%s", card->name);
        snprintf(card->snd_card->longname,sizeof(card->snd_card->longname),
             "%s",card->name);

        //完成 ALSA 层 card 的注册,并创建对应的文件
        ret = snd_card_register(card->snd_card);
        if(ret < 0){
            printk(KERN_ERR "asoc: failed to register soundcard for %s\n",card->name);
            goto probe_dai_err;
        }
```

```
...
        //实例化完成设置标识
        card->instantiated = 1;
        mutex_unlock(&card->mutex);
        return;

probe_dai_err:
        for(i = 0;i < card->num_links;i++)
                soc_remove_dai_link(card,i);

card_probe_error:
        if(card->remove)
                card->remove(pdev);

        snd_card_free(card->snd_card);

        mutex_unlock(&card->mutex);
}
```

从代码分析可见，主要是对 ALSA 层的设备管理部分的封装，通过内部的各个组件的操作来完成整体的注册过程。

对 control 子设备，主要是注册 control 元素；而对 pcm 子设备，则通过 snd_pcm_set_ops 将 pcm 管理实体中 snd_pcm_ops 重新定义为 soc_pcm_ops，具体内容如下：

```
static struct snd_pcm_ops soc_pcm_ops = {
        .open           = soc_pcm_open,
        .close          = soc_codec_close,
        .hw_params      = soc_pcm_hw_params,
        .hw_free        = soc_pcm_hw_free,
        .prepare        = soc_pcm_prepare,
        .trigger        = soc_pcm_trigger,
        .pointer        = soc_pcm_pointer,
};
```

这样上层的操作就可以转入 ALSA SoC 层执行。以 soc_pcm_hw_params 为例来看看具体的操作：

```
static int soc_pcm_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *params)
{
        struct snd_soc_pcm_runtime *rtd = substream->private_data;
        struct snd_soc_platform *platform = rtd->platform;
```

```
struct snd_soc_dai * cpu_dai = rtd -> cpu_dai;
struct snd_soc_dai * codec_dai = rtd -> codec_dai;
int ret =0;

mutex_lock( &pcm_mutex) ;

//物理连接级别的操作,主要进行统一的设置
if( rtd -> dai_link -> ops && rtd -> dai_link -> ops -> hw_params) {
    ret = rtd -> dai_link -> ops -> hw_params( substream,params) ;
    if( ret <0) {
        printk( KERN_ERR "asoc: machine hw_params failed\n" ) ;
        goto out;
    }
}

//codec 侧数据接口的设置
if( codec_dai -> driver -> ops -> hw_params) {
    ret = codec_dai -> driver -> ops -> hw_params( substream,params,codec_dai) ;
    if( ret <0) {
        printk( KERN_ERR "asoc: can't set codec %s hw params\n" ,
            codec_dai -> name) ;
        goto codec_err;
    }
}

//处理器侧数据接口的设置
if( cpu_dai -> driver -> ops -> hw_params) {
    ret = cpu_dai -> driver -> ops -> hw_params( substream,params,cpu_dai) ;
    if( ret <0) {
        printk( KERN_ERR "asoc: interface %s hw params failed\n" ,
            cpu_dai -> name) ;
        goto interface_err;
    }
}

//处理器侧数据操作管理的设置
if( platform -> driver -> ops -> hw_params) {
    ret = platform -> driver -> ops -> hw_params( substream,params) ;
    if( ret <0) {
        printk( KERN_ERR "asoc: platform %s hw params failed\n" ,
            platform -> name) ;
        goto platform_err;
```

```
            }
        }

        rtd->rate = params_rate(params);

out:
        mutex_unlock(&pcm_mutex);
        return ret;

platform_err:
        if(cpu_dai->driver->ops->hw_free)
                cpu_dai->driver->ops->hw_free(substream,cpu_dai);

interface_err:
        if(codec_dai->driver->ops->hw_free)
                codec_dai->driver->ops->hw_free(substream,codec_dai);

codec_err:
        if(rtd->dai_link->ops && rtd->dai_link->ops->hw_free)
                rtd->dai_link->ops->hw_free(substream);

        mutex_unlock(&pcm_mutex);
        return ret;
}
```

可见其主要的功能就是按照一定的顺序对各个组件进行相关的操作，这样整个 pcm 的操作就从上到下打通了。

最后对 SoC 处理器来说，大部分的设备都是以 platform device 出现的，对 ALSA SoC 的架构也要考虑到这一点，其整体上定义为 platform driver，与整个 SoC 系统关联，细节如下：

```
static struct platform_driver soc_driver = {
        .driver         = {
                .name           = "soc-audio",
                .owner          = THIS_MODULE,
                .pm             = &soc_pm_ops,
        },
        .probe          = soc_probe,
        .remove         = soc_remove,
};
```

这样 ALSA SoC 中 machine 层面只要注册对应的 platform device 就可以通过 bus 完成整

体的操作，其中 soc_probe 会试图进行 soc card 的实例化，而每个组件也都可以通过 platform driver 的方式与实际的 SoC 处理器中设备关联，这样在 probe 中注册对应的 ALSA SoC 组件，就可以在底层保证整个物理连接的各个设备最终关联组成上层针对应用的音频设备，这样的设计，既体现了设备层次，也体现了物理连接的实际情况，是一种灵活好用的框架。

这样音频设备框架的整体层次就介绍完了。

6.3.3 音频驱动应用层操作及框架适配

应用程序主要分为控制和 pcm 流的操作两部分。下面以两个例子进行介绍。

首先介绍控制部分的操作，对音频的控制，主要是各个不同通道 mixer 的控制，下面先看看打开的流程：

```
struct mixer * mixer_open(unsigned int card)
{
    struct snd_ctl_elem_list elist;
    struct snd_ctl_elem_info tmp;
    struct snd_ctl_elem_id * eid = NULL;
    struct mixer * mixer = NULL;
    unsigned int n,m;
    int fd;
    char fn[256];

    snprintf(fn,sizeof(fn),"/dev/snd/controlC%u",card);
    //打开相应的设备
    fd = open(fn,O_RDWR);
    if(fd < 0)
        return 0;

    memset(&elist,0,sizeof(elist));
    //首先获得控制信息的个数
    if(ioctl(fd,SNDRV_CTL_IOCTL_ELEM_LIST,&elist) < 0)
        goto fail;

    mixer = calloc(1,sizeof( * mixer));
    if(!mixer)
        goto fail;

    mixer->ctl = calloc(elist.count,sizeof(struct mixer_ctl));
    mixer->info = calloc(elist.count,sizeof(struct snd_ctl_elem_info));
    if(!mixer->ctl || !mixer->info)
        goto fail;
```

```
eid = calloc(elist.count,sizeof(struct snd_ctl_elem_id));
if(!eid)
    goto fail;

mixer->count = elist.count;
mixer->fd = fd;
elist.space = mixer->count;
elist.pids = eid;
//遍历获得各个控制元素的基本信息包括 ID 号等 snd_ctl_elem_id 中的信息
if(ioctl(fd,SNDRV_CTL_IOCTL_ELEM_LIST,&elist) < 0)
    goto fail;

//遍历所有控制元素
for(n = 0;n < mixer->count;n++){
    struct snd_ctl_elem_info * ei = mixer->info + n;
    ei->id.numid = eid[n].numid;
    //获得控制元素的详细信息,主要是值类型范围等.
    if(ioctl(fd,SNDRV_CTL_IOCTL_ELEM_INFO,ei) < 0)
        goto fail;
    mixer->ctl[n].info = ei;
    mixer->ctl[n].mixer = mixer;
    if(ei->type == SNDRV_CTL_ELEM_TYPE_ENUMERATED){
        //如果是则需要继续枚举
        char * * enames = calloc(ei->value.enumerated.items,sizeof(char *));
        if(!enames)
            goto fail;
        mixer->ctl[n].ename = enames;
        for(m = 0;m < ei->value.enumerated.items;m++){
            memset(&tmp,0,sizeof(tmp));
            tmp.id.numid = ei->id.numid;
            tmp.value.enumerated.item = m;
            if(ioctl(fd,SNDRV_CTL_IOCTL_ELEM_INFO,&tmp) < 0)
                goto fail;
            enames[m] = strdup(tmp.value.enumerated.name);
            if(!enames[m])
                goto fail;
        }
    }
}

free(eid);
```

```
        return mixer;

    fail:
        /* TODO: verify frees in failure case */
        if(eid)
            free(eid);
        if(mixer)
            mixer_close(mixer);
        else if(fd >= 0)
            close(fd);
        return 0;
    }
```

从代码中可见，主要的操作就是通过相应的 ioctl 命令获得所有控制元素的信息。对应的还需要读取和写入控制元素指定数据的接口，细节如下：

```
    int mixer_ctl_get_value(struct mixer_ctl *ctl,unsigned int id)
    {
        struct snd_ctl_elem_value ev;
        int ret;
        //控制元素中包含相应的数据
        if(!ctl || (id >= ctl->info->count))
            return -EINVAL;

        memset(&ev,0,sizeof(ev));
        ev.id.numid = ctl->info->id.numid;
        //读取控制元素的所有数据
        ret = ioctl(ctl->mixer->fd,SNDRV_CTL_IOCTL_ELEM_READ,&ev);
        if(ret < 0)
            return ret;

        //根据类型返回相应的值
        switch(ctl->info->type){
        case SNDRV_CTL_ELEM_TYPE_BOOLEAN:
            return !!ev.value.integer.value[id];

        case SNDRV_CTL_ELEM_TYPE_INTEGER:
            return ev.value.integer.value[id];

        case SNDRV_CTL_ELEM_TYPE_ENUMERATED:
            return ev.value.enumerated.item[id];
```

```
        case SNDRV_CTL_ELEM_TYPE_BYTES:
            return ev.value.bytes.data[id];

        default:
            return -EINVAL;
        }

        return 0;
}

int mixer_ctl_set_value(struct mixer_ctl *ctl,unsigned int id,int value)
{
        struct snd_ctl_elem_value ev;
        int ret;
        //控制元素中包含相应的数据
        if(!ctl || (id >= ctl->info->count))
            return -EINVAL;

        memset(&ev,0,sizeof(ev));
        ev.id.numid = ctl->info->id.numid;
        //读取控制元素所有的值
        ret = ioctl(ctl->mixer->fd,SNDRV_CTL_IOCTL_ELEM_READ,&ev);
        if(ret < 0)
            return ret;

        //根据类型修改指定的值
        switch(ctl->info->type){
        case SNDRV_CTL_ELEM_TYPE_BOOLEAN:
            ev.value.integer.value[id] = !!value;
            break;

        case SNDRV_CTL_ELEM_TYPE_INTEGER:
            ev.value.integer.value[id] = value;
            break;

        case SNDRV_CTL_ELEM_TYPE_ENUMERATED:
            ev.value.enumerated.item[id] = value;
            break;

        default:
            return -EINVAL;
        }
```

```
        //将所有值写回
        return ioctl(ctl -> mixer -> fd,SNDRV_CTL_IOCTL_ELEM_WRITE,&ev);
    }
```

从分析中可见，控制元素中的内容读写也是通过 ioctl 来进行的，不过内容是一个整体，在修改的时候需要先读取，修改指定内容后再写回。

了解控制流程后，还需了解 pcm 数据流的操作过程。具体应用层的例子如下：

```
    #include <stdio.h>
    #include <stdlib.h>
    #include "alsa/asoundlib.h"

    int main(int argc,char * argv[])
    {
        int i;
        int ret;
        int buf[128];
        unsigned int val;
        int dir = 0;
        char * buffer;
        int size;
        snd_pcm_uframes_t frames;
        snd_pcm_uframes_t periodsize;
        //pcm 设备句柄
        snd_pcm_t * playback_handle;
        //硬件信息和 pcm 流配置
        snd_pcm_hw_params_t * hw_params;

        FILE * fp = fopen(argv[1],"rb");
        if(fp == NULL)
        return 0;
        fseek(fp,100,SEEK_SET);

        //打开 pcm,最后一个参数为 0 意味着标准配置
        ret = snd_pcm_open(&playback_handle,"default",SND_PCM_STREAM_PLAYBACK,0);

        //分配 snd_pcm_hw_params_t 结构体
        ret = snd_pcm_hw_params_malloc(&hw_params);

        //初始化 hw_params
        ret = snd_pcm_hw_params_any(playback_handle,hw_params);
```

```
    //初始化访问权限
    ret = snd_pcm_hw_params_set_access( playback_handle, hw_params, SND_PCM_ACCESS_RW_IN-
TERLEAVED);

    //初始化采样格式 SND_PCM_FORMAT_U8,8 位
    ret = snd_pcm_hw_params_set_format( playback_handle, hw_params, SND_PCM_FORMAT_U8);

    //设置采样率,如果硬件不支持设置的采样率将使用最接近的
    val = 8000;
    ret = snd_pcm_hw_params_set_rate_near( playback_handle, hw_params, &val, &dir);

    //设置通道数量
    ret = snd_pcm_hw_params_set_channels( playback_handle, hw_params, 2);

    frames = 32;
    periodsize = frames * 2;
    ret = snd_pcm_hw_params_set_buffer_size_near( playback_handle, hw_params, &periodsize);

    periodsize /= 2;
    ret = snd_pcm_hw_params_set_period_size_near( playback_handle, hw_params, &periodsize, 0);

    //设置 hw_params
    ret = snd_pcm_hw_params( playback_handle, hw_params);

    /* Use a buffer large enough to hold one period */
    snd_pcm_hw_params_get_period_size( hw_params, &frames, &dir);

    size = frames * 2; /* 2 bytes/sample, 2 channels */
    buffer = (char *) malloc( size);

    while(1)
    {
        ret = fread( buffer, 1, size, fp);

        //写音频数据到 pcm 设备
        while( ret = snd_pcm_writei( playback_handle, buffer, frames) < 0)
        {
            usleep(2000);
            if( ret == -EPIPE)
            {
                /* EPIPE means underrun */
```

```
                    //完成硬件参数设置,准备好设备
                    snd_pcm_prepare(playback_handle);
                }
                else if(ret < 0)
                {
                    fprintf(stderr,
                        "error from writei: %s\n",
                        snd_strerror(ret));
                }
            }

        }
        //关闭 pcm 设备句柄
        snd_pcm_close(playback_handle);

        return 0;
    }
```

这里的例子代码重点在表述操作流程，移除了相关的错误处理部分，其主要流程还是通过 ALSA 应用层的接口进行操作，主要是打开、设置，然后进行数据操作。

对 Android 框架音频设备的适配，Android 将其作为 HAL 层的模块，相应的模块编译成动态库，运行时由上层框架进行加载。下面以 DM3730 Android 的适配代码为例进行说明。

首先是 HAL 模块说明。

```
    struct audio_module HAL_MODULE_INFO_SYM = {
        .common = {
            .tag = HARDWARE_MODULE_TAG,
            .module_api_version = AUDIO_MODULE_API_VERSION_0_1,
            .hal_api_version = HARDWARE_HAL_API_VERSION,
            .id = AUDIO_HARDWARE_MODULE_ID,
            .name = "Rowboat audio HW HAL",
            .author = "The Android Open Source Project",
            .methods = &hal_module_methods,
        },
    };
```

模块说明中 ID 表示是音频设备的适配模块，操作接口是 methods，其内容如下：

```
    static struct hw_module_methods_t hal_module_methods = {
        .open = adev_open,
    };
```

从中可见主要是 open 操作接口。具体的内容如下：

```
static int adev_open(const hw_module_t * module,const char * name,
                     hw_device_t * * device)
{
    struct audio_device * adev;
    int ret;

    if(strcmp(name,AUDIO_HARDWARE_INTERFACE)!=0)
        return -EINVAL;

    adev = calloc(1,sizeof(struct audio_device));
    if(!adev)
        return -ENOMEM;

    adev->hw_device.common.tag = HARDWARE_DEVICE_TAG;
    adev->hw_device.common.version = AUDIO_DEVICE_API_VERSION_1_0;
    adev->hw_device.common.module = (struct hw_module_t *)module;
    adev->hw_device.common.close = adev_close;

    adev->hw_device.get_supported_devices = adev_get_supported_devices;
    adev->hw_device.init_check = adev_init_check;
    adev->hw_device.set_voice_volume = adev_set_voice_volume;
    adev->hw_device.set_master_volume = adev_set_master_volume;
    adev->hw_device.set_mode = adev_set_mode;
    adev->hw_device.set_mic_mute = adev_set_mic_mute;
    adev->hw_device.get_mic_mute = adev_get_mic_mute;
    adev->hw_device.set_parameters = adev_set_parameters;
    adev->hw_device.get_parameters = adev_get_parameters;
    adev->hw_device.get_input_buffer_size = adev_get_input_buffer_size;
    adev->hw_device.open_output_stream = adev_open_output_stream;
    adev->hw_device.close_output_stream = adev_close_output_stream;
    adev->hw_device.open_input_stream = adev_open_input_stream;
    adev->hw_device.close_input_stream = adev_close_input_stream;
    adev->hw_device.dump = adev_dump;

    adev->ar = audio_route_init();
    adev->orientation = ORIENTATION_UNDEFINED;

*device = &adev->hw_device.common;

    return 0;
}
```

相关的操作就是填充系统需要的接口，这些接口包括控制接口和数据接口。在数据流方面 Android 将音频流中的 playback 和 capture 作为 output stream 和 input stream，这里对应 stream 的 open 和 close 操作。当需要打开时，相应的 stream 会调用 open 接口。以 input stream 操作为例，看一下其详细内容：

```
static int adev_open_input_stream(struct audio_hw_device *dev,
                                  audio_io_handle_t handle,
                                  audio_devices_t devices,
                                  struct audio_config *config,
                                  struct audio_stream_in **stream_in)
{
    struct audio_device *adev = (struct audio_device *)dev;
    struct stream_in *in;
    int ret;

    *stream_in = NULL;

    /* Respond with a request for mono if a different format is given. */
    if(config->channel_mask != AUDIO_CHANNEL_IN_MONO) {
        config->channel_mask = AUDIO_CHANNEL_IN_MONO;
        return -EINVAL;
    }

    in = (struct stream_in *)calloc(1,sizeof(struct stream_in));
    if(!in)
        return -ENOMEM;

    in->stream.common.get_sample_rate = in_get_sample_rate;
    in->stream.common.set_sample_rate = in_set_sample_rate;
    in->stream.common.get_buffer_size = in_get_buffer_size;
    in->stream.common.get_channels = in_get_channels;
    in->stream.common.get_format = in_get_format;
    in->stream.common.set_format = in_set_format;
    in->stream.common.standby = in_standby;
    in->stream.common.dump = in_dump;
    in->stream.common.set_parameters = in_set_parameters;
    in->stream.common.get_parameters = in_get_parameters;
    in->stream.common.add_audio_effect = in_add_audio_effect;
    in->stream.common.remove_audio_effect = in_remove_audio_effect;
    in->stream.set_gain = in_set_gain;
    in->stream.read = in_read;
    in->stream.get_input_frames_lost = in_get_input_frames_lost;
```

```
        in->dev = adev;
        in->standby = true;
        in->requested_rate = config->sample_rate;
        in->pcm_config = &pcm_config_in;/* default PCM config */

        *stream_in = &in->stream;
        return 0;
}
```

其中仍是接口的设置，但可见其中的设置基本与 pcm 相关的参数一致，实际的适配就是完成以上的接口函数。

以上具体的接口无论是控制接口还是数据接口都会通过 Android 提供 Tiny ALSA 来实现相应的功能，Tiny ALSA 实际是 ALSA 应用层库的简化。适配本身就是按照设备具体的情况进行控制流和数据流接口的实现。

6.3.4 TI 芯片音频驱动相关实现详解

实际音频驱动的分析会以 DM3730 的相关驱动为例进行说明。主要以 ASoC 中的各个模块进行介绍。

1. machine 模块

首先来看一下初始化部分：

```
//machine 相关的初始化函数
static int __init omap3evm_soc_init(void)
{
        int ret;

        if(!machine_is_omap3evm())
                return -ENODEV;
        pr_info("OMAP3 EVM SoC init\n");

        //这里分配并注册 platform_device 是因为 ASoC core 中有
        //soc-audio driver,其中 probe 会注册 soc_card
        omap3evm_snd_device = platform_device_alloc("soc-audio", -1);
        if(!omap3evm_snd_device){
                printk(KERN_ERR "Platform device allocation failed\n");
                return -ENOMEM;
        }

        //设置 soc_card 并加入 platform_deivce,以便 soc-audio
```

```
        //进行实际的注册 soc_card 操作
        platform_set_drvdata(omap3evm_snd_device,&snd_soc_omap3evm);
        ret = platform_device_add(omap3evm_snd_device);
        if(ret)
            goto err1;

        return 0;

err1:
        printk(KERN_ERR "Unable to add platform device\n");
        platform_device_put(omap3evm_snd_device);

        return ret;
}
```

其中涉及的主要部分就是 snd_soc_omap3evm，包含了物理连接的属性，具体内容如下：

```
//具体的 CPU dai/codec/codec dai/platform 之间的 link 说明
//说明具体用哪些 CPU dai/codec/codec dai/platform.
static struct snd_soc_dai_link omap3evm_dai[] = {
{
        .name           = "TWL4030",
        .stream_name    = "TWL4030",
        //用 omap3 mcbsp2
        .cpu_dai_name   = "omap-mcbsp-dai.1",
        .codec_dai_name = "twl4030-hifi",
        .platform_name  = "omap-pcm-audio",
        .codec_name     = "twl4030-codec",
        .ops            = &omap3evm_ops,
},
};

/* Audio machine driver */
//soc card 的说明,最后在 ALSA 中为 card
static struct snd_soc_card snd_soc_omap3evm = {
        .name = "omap3evm",
        .dai_link = omap3evm_dai,
        .num_links = ARRAY_SIZE(omap3evm_dai),
};
```

操作接口 omap3evm_ops 主要就是硬件参数的接口，详细分析如下：

```
//该接口主要是 pcm 使用,用于设置 machine 的 CPU dai,codec dai 的参数并保持一致.
static int omap3evm_hw_params(struct snd_pcm_substream * substream,
    struct snd_pcm_hw_params * params)
{
    struct snd_soc_pcm_runtime * rtd = substream -> private_data;
    struct snd_soc_dai * codec_dai = rtd -> codec_dai;
    struct snd_soc_dai * cpu_dai = rtd -> cpu_dai;
    int ret;

    /* Set codec DAI configuration */
    //设置 codec dai 为主 I2S 格式
    ret = snd_soc_dai_set_fmt(codec_dai,SND_SOC_DAIFMT_I2S |
        SND_SOC_DAIFMT_NB_NF | SND_SOC_DAIFMT_CBM_CFM);
    if(ret < 0) {
        printk(KERN_ERR "Can't set codec DAI configuration\n");
        return ret;
    }

    /* Set cpu DAI configuration */
    //设置 CPU dai 为从 I2S 格式
    ret = snd_soc_dai_set_fmt(cpu_dai,SND_SOC_DAIFMT_I2S
         | SND_SOC_DAIFMT_NB_NF | SND_SOC_DAIFMT_CBM_CFM);
    if(ret < 0) {
        printk(KERN_ERR "Can't set cpu DAI configuration\n");
        return ret;
    }

    /* Set the codec system clock for DAC and ADC */
    //设置 clks
    ret = snd_soc_dai_set_sysclk(codec_dai,0,26000000,SND_SOC_CLOCK_IN);
    if(ret < 0) {
        printk(KERN_ERR "Can't set codec system clock\n");
        return ret;
    }

    return 0;
}
```

从整体上来看 machine 部分主要负责整体连接的设定，并不涉及太多的操作，具体的操作也是在整体上保证系统中各个接口的一致性。而 ALSA 框架为了保证这种一致性也分别提供了对处理器侧 dai 接口以及 codec 侧 dai 接口进行操作的函数，machine 模块就是通过这些函数来完成对应的操作的。

2. platform 模块

该模块主要是使用 DMA 进行数据传输，下面进行具体分析。

首先该模块要提供 snd_soc_platform_driver，并在相应的 platform device 被 probe 时注册，详细的信息如下：

```
//ASoC platform 接口主要是 CPU DMA 的接口
static struct snd_soc_platform_driver omap_soc_platform = {
        .ops            = &omap_pcm_ops,
        .pcm_new        = omap_pcm_new,
        .pcm_free       = omap_pcm_free_dma_buffers,
};
```

主要的部分是各种操作接口 omap_pcm_ops 以及创建 pcm 时初始化的 pcm_new 接口 omap_pcm_new。对于 omap_pcm_new 的细节如下：

```
//创建 pcm 的接口,DMA 相关的 new 操作主要是为 stream 分配 DMA buffer 空间
static int omap_pcm_new(struct snd_card *card,struct snd_soc_dai *dai,
                struct snd_pcm *pcm)
{
        int ret = 0;

        if(!card->dev->dma_mask)
                card->dev->dma_mask = &omap_pcm_dmamask;
        if(!card->dev->coherent_dma_mask)
                card->dev->coherent_dma_mask = DMA_BIT_MASK(64);

        //为相应的 playback/capture stream 分配 DMA buffer
        if(dai->driver->playback.channels_min){
        ret = omap_pcm_preallocate_dma_buffer(pcm,SNDRV_PCM_STREAM_PLAYBACK);
                if(ret)
                        goto out;
        }

        if(dai->driver->capture.channels_min){
        ret = omap_pcm_preallocate_dma_buffer(pcm,SNDRV_PCM_STREAM_CAPTURE);
                if(ret)
                        goto out;
        }


out:
        return ret;
}
```

主要是分配空间，而 omap_pcm_ops 的内容如下：

```
//pcm 的操作接口
static struct snd_pcm_ops omap_pcm_ops = {
        .open           = omap_pcm_open,
        .close          = omap_pcm_close,
        .ioctl          = snd_pcm_lib_ioctl,
        .hw_params      = omap_pcm_hw_params,
        .hw_free        = omap_pcm_hw_free,
        .prepare        = omap_pcm_prepare,
        .trigger        = omap_pcm_trigger,
        .pointer        = omap_pcm_pointer,
        .mmap           = omap_pcm_mmap,
};
```

这里主要是各种操作接口，下面对几个接口进行分析：

```
//pcm open 是 DMA 的接口,主要分配驱动相关的 runtime 的管理实体
//包括 DMA channel 和 DMA 属性等
static int omap_pcm_open(struct snd_pcm_substream *substream)
{
        struct snd_pcm_runtime *runtime = substream->runtime;
        struct omap_runtime_data *prtd;
        int ret;

        //设置 pcm_hardware 限制
        snd_soc_set_runtime_hwparams(substream,&omap_pcm_hardware);

        /* Ensure that buffer size is a multiple of period size */
        //设置 periods 必须为整数的限制
        ret = snd_pcm_hw_constraint_integer(runtime,SNDRV_PCM_HW_PARAM_PERIODS);
        if(ret < 0)
                goto out;

        //分配相应的 runtime 管理实体并初始化
        prtd = kzalloc(sizeof(*prtd),GFP_KERNEL);
        if(prtd == NULL){
                ret = -ENOMEM;
                goto out;
        }

        //初始化相应的 spin lock
        spin_lock_init(&prtd->lock);
```

```
	runtime -> private_data = prtd;

out:
	return ret;
}

//pcm 设置 hw 参数的接口,主要是申请 DMA,这里保证 DMA 的设置
//和 dai 具体接口的设置分开,保证 DMA 设置的独立性.
static int omap_pcm_hw_params(struct snd_pcm_substream * substream,
			struct snd_pcm_hw_params * params)
{
	struct snd_pcm_runtime * runtime = substream -> runtime;
	struct snd_soc_pcm_runtime * rtd = substream -> private_data;
	struct omap_runtime_data * prtd = runtime -> private_data;
	struct omap_pcm_dma_data * dma_data;

	int err = 0;

	//该函数要将 dai 的 DMA 属性转换为 pcm runtime 使用的 DMA 属性
	//dai 管理的通常是物理 DMA 属性信息,runtime 管理的是动态
	//分配的 logical channel 等信息.

	//首先获得 CPU dai 设置的 pcm 所带的 DMA 管理信息
	//相关的信息是在 CPU dai 的 hw_params 中设置的,
	//按照 soc - core 的执行逻辑先执行 CPU dai 的 hw_params
	//然后执行 DMA 的 hw_params,dai 的 hw_params 主要是进行
	//基本的传输属性的设置包括 dai 使用的 dma_req 信号
	//地址、sync 方式等
	dma_data = snd_soc_dai_get_dma_data(rtd -> cpu_dai,substream);

	/ * return if this is a bufferless transfer e. g.
	 * codec < --> BT codec or GSM modem -- lg FIXME * /
	if(!dma_data)
		return 0;

	//设置 substream 的 DMA buffer,这些 buffer 应该是在 pcm_new 时分配的
	snd_pcm_set_runtime_buffer(substream,&substream -> dma_buffer);

	//设置 pcm 流的 DMA buffer 占用的 bytes,为对应用层映射 mmap 准备
	//在 dai 的 dai_startup 接口中会对 SNDRV_PCM_HW_PARAM_BUFFER_SIZE
	//对于 dai 的 fifo size 进行修正从而保证这里有合适的 size 值。dai_startup 会
	//在 pcm_open 中调用。而在 hw_params 的时候会根据之前 pcm_open 时 add 的规则
```

```
        //重新提炼传入的参数保证符合硬件的属性,这是应该会调整相应的值.
        runtime->dma_bytes = params_buffer_bytes(params);

        if(prtd->dma_data)
                return 0;
        //将 dai 管理的 DMA 属性关联到 runtime 管理的属性中
        prtd->dma_data = dma_data;
        //请求 DMA channel
        err = omap_request_dma(dma_data->dma_req,dma_data->name,
                        omap_pcm_dma_irq,substream,&prtd->dma_ch);
        if(!err){
                /*
                 * Link channel with itself so DMA doesn't need any
                 * reprogramming while looping the buffer
                 */
                //DMA channel link 到自己,保证 DMA 的持续传输.
                //这里只是设置 link,真正 link 的 enable 要在 DMA start 才开始.
                omap_dma_link_lch(prtd->dma_ch,prtd->dma_ch);
        }

        return err;
}

//对应于 pcm 的 trigger 接口,主要用来控制状态 start stop suspend resume 等
static int omap_pcm_trigger(struct snd_pcm_substream * substream,int cmd)
{
        struct snd_pcm_runtime * runtime = substream->runtime;
        struct omap_runtime_data * prtd = runtime->private_data;
        struct omap_pcm_dma_data * dma_data = prtd->dma_data;
        unsigned long flags;
        int ret = 0;

        spin_lock_irqsave(&prtd->lock,flags);
        switch(cmd){
        case SNDRV_PCM_TRIGGER_START:
        case SNDRV_PCM_TRIGGER_RESUME:
        case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
                prtd->period_index = 0;
                /* Configure McBSP internal buffer usage */
                //DMA 开始时要保证可以触发 DMA request,所以有必要设置 threshold
                if(dma_data->set_threshold)
                        dma_data->set_threshold(substream);
```

```
            //enable 相应的 DMA 及其 irq
            omap_start_dma(prtd -> dma_ch);
            break;

        case SNDRV_PCM_TRIGGER_STOP:
        case SNDRV_PCM_TRIGGER_SUSPEND:
        case SNDRV_PCM_TRIGGER_PAUSE_PUSH:
            //停止相关的操作,要 stop DMA
            prtd -> period_index = -1;
            omap_stop_dma(prtd -> dma_ch);
            break;
        default:
            ret = -EINVAL;
        }
        spin_unlock_irqrestore(&prtd -> lock,flags);

        return ret;
    }
```

其具体的细节就是根据需要通过 ALSA pcm 接口设置参数限制，另外主要的工作就是在相应的操作接口对 DMA 进行设置和正确地操作。

3. 处理器侧 dai 模块

DM3730 中音频流的接口设备是 McBSP（multichannel buffered serial port），其可以支持各种音频流的传输方式如（I^2S、TCM）等，实际的代码中就是要根据参数的格式对硬件进行正确的设置。对于硬件 McBSP 的框架结构如图 6-17 所示。

图 6-17 引自《DM3730 芯片手册》中第 3096 页框图。在 dai 模块中主要通过设置其中左上的寄存器组来完成，而 DMA 则是通过操作 DRR 和 DXR 寄存器来完成数据的读写。

下面来看看 dai 相关的实现细节。对于驱动主要是注册 dai 接口的驱动，其细节如下：

```
    //CPU dai 的 driver 说明
    static struct snd_soc_dai_driver omap_mcbsp_dai =
    {
        .probe = mcbsp_dai_probe,
        .playback = {
            //playback 的能力说明
            .channels_min = 1,
            .channels_max = 16,
            .rates = OMAP_MCBSP_RATES,
            .formats = SNDRV_PCM_FMTBIT_S16_LE | SNDRV_PCM_FMTBIT_S32_LE,
        },
```

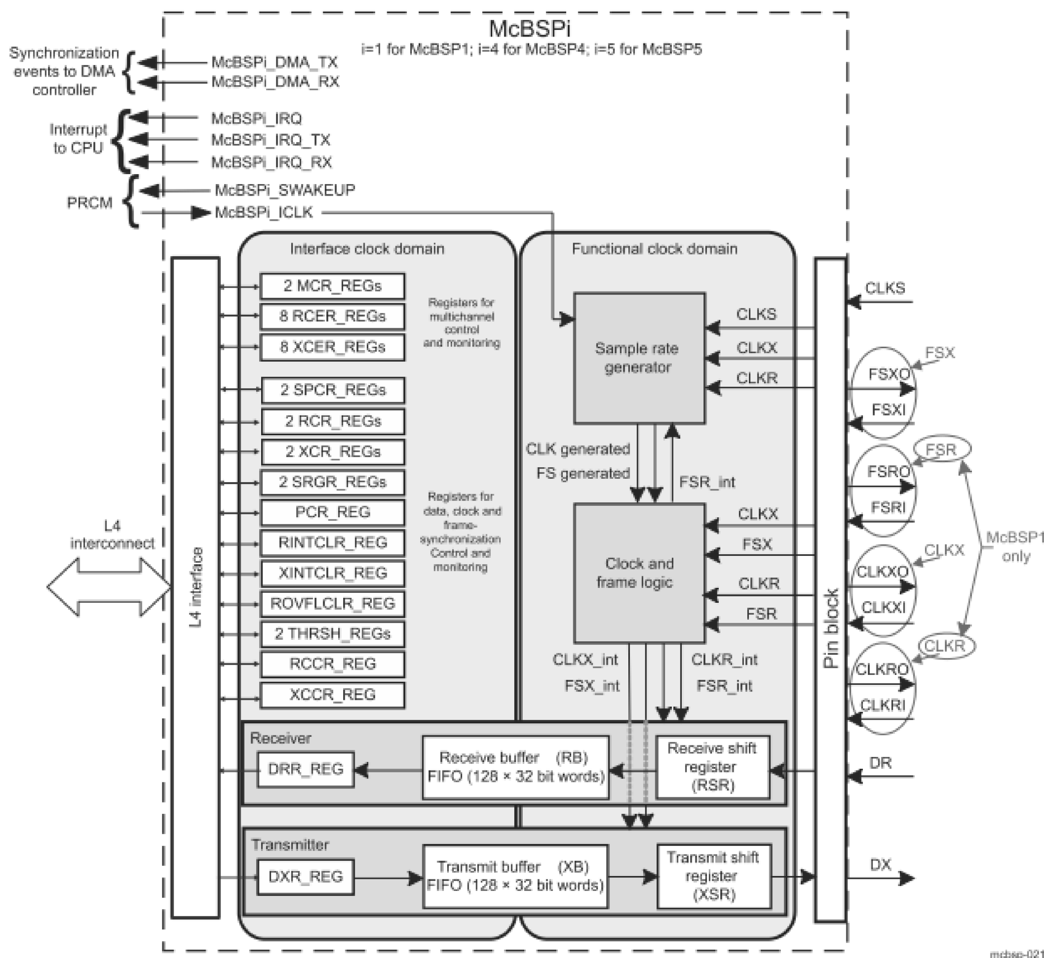


图 6-17　McBSP 框架

```
        .capture = {
            //capture 的能力说明
            .channels_min = 1,
            .channels_max = 16,
            .rates = OMAP_MCBSP_RATES,
            .formats = SNDRV_PCM_FMTBIT_S16_LE | SNDRV_PCM_FMTBIT_S32_LE,
        },
        //操作接口
        .ops = &mcbsp_dai_ops,
    };
```

可见主要包括数据能力和操作接口。操作接口的细节如下：

```
    //这里是重要的 dai_ops 集合
    static struct snd_soc_dai_ops mcbsp_dai_ops = {
```

```
        .startup        = omap_mcbsp_dai_startup,
        .shutdown       = omap_mcbsp_dai_shutdown,
        .trigger        = omap_mcbsp_dai_trigger,
        .delay          = omap_mcbsp_dai_delay,
        .hw_params      = omap_mcbsp_dai_hw_params,
        .set_fmt        = omap_mcbsp_dai_set_dai_fmt,
        .set_clkdiv     = omap_mcbsp_dai_set_clkdiv,
        .set_sysclk     = omap_mcbsp_dai_set_dai_sysclk,
};
```

下面对主要的接口进行分析：

```
//pcm 下进行 hw 参数设置的接口,其中的操作主要是:1. 设置 McBSP 相关
//的 DMA 属性,为 CPU DMA 提供相应的配置数据;2. 进行 McBSP
//接口的必要寄存器设置,这里主要是对 word length 和 frame size 涉及音频
//应用层参数的最后设置,加上之前 dai 的 set_dai_fmt 的 format 基本设置后
//统一通过 plat-omap 下 McBSP 提供的 config 接口设置寄存器
static int omap_mcbsp_dai_hw_params(struct snd_pcm_substream *substream,
                                    struct snd_pcm_hw_params *params,
                                    struct snd_soc_dai *cpu_dai)
{
    struct omap_mcbsp_data *mcbsp_data = snd_soc_dai_get_drvdata(cpu_dai);
    struct omap_mcbsp_reg_cfg *regs = &mcbsp_data->regs;
    struct omap_pcm_dma_data *dma_data;
    int dma,bus_id = mcbsp_data->bus_id;
    int wlen,channels,wpf,sync_mode = OMAP_DMA_SYNC_ELEMENT;
    int pkt_size = 0;
    unsigned long port;
    unsigned int format,div,framesize,master;

    //首先获得基本的 McBSP 的 DMA 的配置静态空间
    dma_data = &omap_mcbsp_dai_dma_params[cpu_dai->id][substream->stream];
    if(cpu_class_is_omap1()){
        dma = omap1_dma_reqs[bus_id][substream->stream];
        port = omap1_mcbsp_port[bus_id][substream->stream];
    }else if(cpu_is_omap2420()){
        dma = omap24xx_dma_reqs[bus_id][substream->stream];
        port = omap2420_mcbsp_port[bus_id][substream->stream];
    }else if(cpu_is_omap2430()){
        dma = omap24xx_dma_reqs[bus_id][substream->stream];
        port = omap2430_mcbsp_port[bus_id][substream->stream];
    }else if(cpu_is_omap343x()){
```

```
        dma = omap24xx_dma_reqs[bus_id][substream -> stream];
        port = omap34xx_mcbsp_port[bus_id][substream -> stream];
    } else {
        return - ENODEV;
    }
    //设置 DMA 的 word length
    switch(params_format(params)) {
    case SNDRV_PCM_FORMAT_S16_LE:
        dma_data -> data_type = OMAP_DMA_DATA_TYPE_S16;
        wlen = 16;
        break;
    case SNDRV_PCM_FORMAT_S32_LE:
        dma_data -> data_type = OMAP_DMA_DATA_TYPE_S32;
        wlen = 32;
        break;
    default:
        return - EINVAL;
    }
    if(cpu_is_omap343x()) {
        //这里提供给 CPU DMA 进行 dai threshold 设置的接口,保证 DMA 状态
        //发生变化特别是 start 时 threshold 应该进行相应的设置.
        //CPU DMA 的 trigger 接口会调用该接口进行操作.
        dma_data -> set_threshold = omap_mcbsp_set_threshold;
        /* TODO: Currently, MODE_ELEMENT == MODE_FRAME */
        if(omap_mcbsp_get_dma_op_mode(bus_id)
          == MCBSP_DMA_MODE_THRESHOLD) {
            int period_words, max_thrsh;

            period_words = params_period_bytes(params)/(wlen/8);
            if(substream -> stream == SNDRV_PCM_STREAM_PLAYBACK)
                max_thrsh = omap_mcbsp_get_max_tx_threshold(
                                mcbsp_data -> bus_id);
            else
                max_thrsh = omap_mcbsp_get_max_rx_threshold(
                                mcbsp_data -> bus_id);
            /*
             * If the period contains less or equal number of words,
             * we are using the original threshold mode setup:
             * McBSP threshold = sDMA frame size = period_size
             * Otherwise we switch to sDMA packet mode:
             * McBSP threshold = sDMA packet size
             * sDMA frame size = period size
```

```
            */
            //如果一个 period 包含的 McBSP word 超过了阈值,会有不必要的
            //DMA 和 dai 同步的问题主要是硬件接口要求一次传输为 fifo 阈值
            //而逻辑上无法满足 DMA 一个中断传输 period 大小的数据。这就要求
            //进行 DMA packet sync 操作修正相应的 packet 值,保证 DMA 的正确传输和
            //正确中断.
            if(period_words > max_thrsh){
                int divider = 0;

                /*
                 * Look for the biggest threshold value,which
                 * divides the period size evenly.
                 */
                divider = period_words/max_thrsh;
                if(period_words % max_thrsh)
                    divider++;
                while(period_words % divider &&
                    divider < period_words)
                    divider++;
                if(divider == period_words)
                    return -EINVAL;

                pkt_size = period_words/divider;
                sync_mode = OMAP_DMA_SYNC_PACKET;
            }else{
                //这里 period 包含的数据大小小于 threshold
                //即 frame size 满足 fifo threshold,只要设置 sync_frame 即可
                sync_mode = OMAP_DMA_SYNC_FRAME;
            }
        }
    }

    //DMA 的其他属性设置
    dma_data->name = substream->stream ? "Audio Capture" : "Audio Playback";
    dma_data->dma_req = dma;
    dma_data->port_addr = port;
    //下面和 hardware sync mode 相关
    dma_data->sync_mode = sync_mode;
    dma_data->packet_size = pkt_size;

    //设置 cpu_dai 的 DMA 属性,以便后续 CPU DMA hw_param 使用.
    snd_soc_dai_set_dma_data(cpu_dai,substream,dma_data);
```

```
//相应的 McBSP 接口必须首次设置,否则直接返回
if(mcbsp_data->configured){
    /* McBSP already configured by another stream */
    return 0;
}

//下面是基本的 format 设置,主要包括 word、frame 和 phase 的设置
//另外注意对于 format 的整体设置包括下面几个部分
//SND_SOC_DAIFMT_FORMAT_MASK;SND_SOC_DAIFMT_CLOCK_MASK
//SND_SOC_DAIFMT_INV_MASK;SND_SOC_DAIFMT_MASTER_MASK
format = mcbsp_data->fmt & SND_SOC_DAIFMT_FORMAT_MASK;
wpf = channels = params_channels(params);
if(channels == 2 &&(format == SND_SOC_DAIFMT_I2S ||
            format == SND_SOC_DAIFMT_LEFT_J)){
    /* Use dual-phase frames */
    //如果是 I²S 格式,则设置 dual-phase
    regs->rcr2      |= RPHASE;
    regs->xcr2      |= XPHASE;
    /* Set 1 word per(McBSP)frame for phase1 and phase2 */
    //注意 dual-phase 模式设置 frame 的 channel 必须为 1
    //但是 word size 可以两个 phase 不同
    wpf--;
    regs->rcr2      |= RFRLEN2(wpf - 1);
    regs->xcr2      |= XFRLEN2(wpf - 1);
}

//设置 frame 的 channel 数
regs->rcr1          |= RFRLEN1(wpf - 1);
regs->xcr1          |= XFRLEN1(wpf - 1);

//设置 word length
switch(params_format(params)){
case SNDRV_PCM_FORMAT_S16_LE:
    /* Set word lengths */
    regs->rcr2      |= RWDLEN2(OMAP_MCBSP_WORD_16);
    regs->rcr1      |= RWDLEN1(OMAP_MCBSP_WORD_16);
    regs->xcr2      |= XWDLEN2(OMAP_MCBSP_WORD_16);
    regs->xcr1      |= XWDLEN1(OMAP_MCBSP_WORD_16);
    break;
case SNDRV_PCM_FORMAT_S32_LE:
```

```
        /* Set word lengths */
        regs->rcr2      |= RWDLEN2(OMAP_MCBSP_WORD_32);
        regs->rcr1      |= RWDLEN1(OMAP_MCBSP_WORD_32);
        regs->xcr2      |= XWDLEN2(OMAP_MCBSP_WORD_32);
        regs->xcr1      |= XWDLEN1(OMAP_MCBSP_WORD_32);
        break;
default:
        /* Unsupported PCM format */
        return -EINVAL;
}

/* In McBSP master modes, FRAME(i.e. sample rate) is generated
 * by _counting_ BCLKs. Calculate frame size in BCLKs */
//检查是 master 还是 slave,如果 codec 的 bit clock 和 frame sync 都做 slave 即
//SND_SOC_DAIFMT_CBS_CFS 则 McBSP 做 master
master = mcbsp_data->fmt & SND_SOC_DAIFMT_MASTER_MASK;
//下面 frame  size 的值主要是看占多少 bit clocks
if(master == SND_SOC_DAIFMT_CBS_CFS){
        //当 McBSP 做 master 的时候,frame size 应该通过 bit clock 计算
        //div 为分频的除数
        div = mcbsp_data->clk_div ? mcbsp_data->clk_div : 1;
        //根据采样率算出 frame 占多少 bit clock
        framesize = (mcbsp_data->in_freq/div)/params_rate(params);

        if(framesize < wlen * channels){
                printk(KERN_ERR "%s: not enough bandwidth for desired rate and "
                                "channels\n",__func__);
                return -EINVAL;
        }
}else
        //slave 设备 frame size 直接 word length × channel 数目即可
        framesize = wlen * channels;

//至此 frame size 计算出来要根据相应的值填 FS 信号间隔周期
//以及 FS 信号的宽度。相应的寄存器值+1 才是实际的
//周期,所以填写寄存器的时候都-1.
/* Set FS period and length in terms of bit clock periods */
switch(format){
case SND_SOC_DAIFMT_I2S:
case SND_SOC_DAIFMT_LEFT_J:
        //fs 周期间隔 frame size
```

```
        regs->srgr2        |= FPER(framesize - 1);
        //这里是 2 phase mode,所以 fs 宽度为 frame size/2
        regs->srgr1        |= FWID((framesize >> 1) - 1);
        break;
    case SND_SOC_DAIFMT_DSP_A:
    case SND_SOC_DAIFMT_DSP_B:
        //fs 周期间隔 frame size
        regs->srgr2        |= FPER(framesize - 1);
        //fs 宽度只要 1 bit clock 即可
        regs->srgr1        |= FWID(0);
        break;
    }

    //至此 pcm 的 hw_params 调用已经进行的所有的 McBSP 接口
    //的配置,所以可以进行实际的 McBSP 寄存器的 config 操作
    omap_mcbsp_config(bus_id, &mcbsp_data->regs);
    //记录 word length 并标记已经进行实际硬件接口 config
    mcbsp_data->wlen = wlen;
    mcbsp_data->configured = 1;

    return 0;
}

//下面的函数主要是对与 dai 接口设置 format,应该和 codec dai 的 format 一致.
//这里 format 主要是基本的音频传输协议格式包括下面几个方面:
//SND_SOC_DAIFMT_FORMAT_MASK 基本格式比如 I2S pcm 等
//SND_SOC_DAIFMT_CLOCK_MASK clock 设置
//SND_SOC_DAIFMT_INV_MASK 信号高低有效主要是极性设置
//SND_SOC_DAIFMT_MASTER_MASK 主从设置
static int omap_mcbsp_dai_set_dai_fmt(struct snd_soc_dai *cpu_dai,
                                      unsigned int fmt)
{
    struct omap_mcbsp_data *mcbsp_data = snd_soc_dai_get_drvdata(cpu_dai);
    struct omap_mcbsp_reg_cfg *regs = &mcbsp_data->regs;
    unsigned int temp_fmt = fmt;

    if(mcbsp_data->configured)
        return 0;

    //保存上层的 format 设置到 McBSP 的管理结构中
    mcbsp_data->fmt = fmt;
```

```
memset(regs,0,sizeof(*regs));
/* Generic McBSP register settings */
//基本的 McBSP 配置,设置发送 sync error 中断
//并且让 McBSP 一直运行
regs->spcr2          |= XINTM(3) | FREE;
//设置接收 receive sync error 中断
regs->spcr1          |= RINTM(3);
/* RFIG and XFIG are not defined in 34xx */
if(!cpu_is_omap34xx()){
    regs->rcr2       |= RFIG;
    regs->xcr2       |= XFIG;
}
if(cpu_is_omap2430() || cpu_is_omap34xx()){
    //设置 DXENDLY - When McBSPi. MCBSPLP_SPCR1_REG[7] DXENA bit is
    //set to one,this field selects the added delay
    //XDMAEN - Transmit DMA Enable bit 主要是 DMA req 信号的使能
    //XDISABLE - disable transmit
    regs->xccr = DXENDLY(1) | XDMAEN | XDISABLE;
    //设置 receive 的 full_cycle 表示在 fs 和 data 在 bclk 的相同边沿采样
    //RDMAEN - Receive DMA Enable bit 主要是 DMA req 信号的使能
    //RDISABLE - disable receive
    regs->rccr = RFULL_CYCLE | RDMAEN | RDISABLE;
}

//根据要设置的 format 进行基本的 fs 后的 data 延时设置
//以及信号极性 bclk 采样和 fs 极性设置
//Normal BCLK + FS: FS active low. TX data driven on falling edge of bit clock
//and RX data sampled on rising edge of bit clock.
//对于音频 format 的详细信息见 trm21.2.4
switch(fmt & SND_SOC_DAIFMT_FORMAT_MASK){
case SND_SOC_DAIFMT_I2S:
    /* 1-bit data delay */
    regs->rcr2      |= RDATDLY(1);
    regs->xcr2      |= XDATDLY(1);
    break;
case SND_SOC_DAIFMT_LEFT_J:
    /* 0-bit data delay */
    regs->rcr2      |= RDATDLY(0);
    regs->xcr2      |= XDATDLY(0);
    regs->spcr1     |= RJUST(2);
    /* Invert FS polarity configuration */
```

```
        //fs 信号极性反
        temp_fmt ^= SND_SOC_DAIFMT_NB_IF;
        break;
    case SND_SOC_DAIFMT_DSP_A:
        /* 1-bit data delay */
        regs->rcr2          |= RDATDLY(1);
        regs->xcr2          |= XDATDLY(1);
        /* Invert FS polarity configuration */
        //fs 信号极性反
        temp_fmt ^= SND_SOC_DAIFMT_NB_IF;
        break;
    case SND_SOC_DAIFMT_DSP_B:
        /* 0-bit data delay */
        regs->rcr2          |= RDATDLY(0);
        regs->xcr2          |= XDATDLY(0);
        /* Invert FS polarity configuration */
        //fs 信号极性反
        temp_fmt ^= SND_SOC_DAIFMT_NB_IF;
        break;
    default:
        /* Unsupported data format */
        return -EINVAL;
    }

    //做 master/slave 的设置
    switch(fmt & SND_SOC_DAIFMT_MASTER_MASK) {
    case SND_SOC_DAIFMT_CBS_CFS:
        /* McBSP master. Set FS and bit clocks as outputs */
        //做 master,fs 和 clk 信号输出
        regs->pcr0          |= FSXM | FSRM |
                    CLKXM | CLKRM;
        /* Sample rate generator drives the FS */
        //产生采样信号
        regs->srgr2         |= FSGM;
        break;
    case SND_SOC_DAIFMT_CBM_CFM:
        /* McBSP slave */
        break;
    default:
        /* Unsupported master/slave configuration */
        return -EINVAL;
```

```
    }

    /* Set bit clock(CLKX/CLKR)and FS polarities */
    //设置 bclk 信号和 fs 信号极性
    switch(temp_fmt & SND_SOC_DAIFMT_INV_MASK){
    case SND_SOC_DAIFMT_NB_NF:
        /*
         * Normal BCLK + FS.
         * FS active low. TX data driven on falling edge of bit clock
         * and RX data sampled on rising edge of bit clock.
         */
        //寄存器 fs 和 clk 极性设置都设为 nb + nf
        regs->pcr0      |= FSXP | FSRP |
                CLKXP | CLKRP;
        break;
    case SND_SOC_DAIFMT_NB_IF:
        //clk 极性设置设了为 nb
        regs->pcr0      |= CLKXP | CLKRP;
        break;
    case SND_SOC_DAIFMT_IB_NF:
        //fs 极性设置设了为 nf
        regs->pcr0      |= FSXP | FSRP;
        break;
    case SND_SOC_DAIFMT_IB_IF:
        //都不设为 ib + if
        break;
    default:
        return -EINVAL;
    }

    return 0;
}

//下面的函数主要是对与 dai 接口设置 format,应该和 codec dai 的 format 一致.
//这里 format 主要是基本的音频传输协议格式包括下面几个方面
//SND_SOC_DAIFMT_FORMAT_MASK 基本格式比如 I2S pcm 等
//SND_SOC_DAIFMT_CLOCK_MASK clock 设置
//SND_SOC_DAIFMT_INV_MASK 信号高低有效主要是极性设置
//SND_SOC_DAIFMT_MASTER_MASK 主从设置
static int omap_mcbsp_dai_set_dai_fmt(struct snd_soc_dai *cpu_dai,
                unsigned int fmt)
{
```

```
struct omap_mcbsp_data * mcbsp_data = snd_soc_dai_get_drvdata(cpu_dai);
struct omap_mcbsp_reg_cfg * regs = &mcbsp_data -> regs;
unsigned int temp_fmt = fmt;

if(mcbsp_data -> configured)
    return 0;

//保存上层的 format 设置到 McBSP 的管理结构中
mcbsp_data -> fmt = fmt;
memset(regs,0,sizeof( * regs));
/ * Generic McBSP register settings * /
//基本的 McBSP 配置,设置发送 sync error 中断
//并且让 McBSP 一直运行
regs -> spcr2          |= XINTM(3) | FREE;
//设置接收 receive sync error 中断
regs -> spcr1          |= RINTM(3);
/ * RFIG and XFIG are not defined in 34xx * /
if(!cpu_is_omap34xx()){
    regs -> rcr2       |= RFIG;
    regs -> xcr2       |= XFIG;
}
if(cpu_is_omap2430() || cpu_is_omap34xx()){
    //设置 DXENDLY - When McBSPi. MCBSPLP_SPCR1_REG[7] DXENA bit is
    //set to one,this field selects the added delay
    //XDMAEN - Transmit DMA Enable bit 主要是 DMA req 信号的使能
    //XDISABLE - disable transmit
    regs -> xccr = DXENDLY(1) | XDMAEN | XDISABLE;
    //设置 receive 的 full_cycle 表示在 fs 和 data 在 bclk 的相同边沿采样
    //RDMAEN - Receive DMA Enable bit 主要是 DMA req 信号的使能
    //RDISABLE - disable receive
    regs -> rccr = RFULL_CYCLE | RDMAEN | RDISABLE;
}

//根据要设置的 format 进行基本的 fs 后的 data 延时设置
//以及信号极性 bclk 采样和 fs 极性设置
//Normal BCLK + FS: FS active low. TX data driven on falling edge of bit clock
//and RX data sampled on rising edge of bit clock.
//对于音频 format 的详细信息见 trm21.2.4
switch(fmt & SND_SOC_DAIFMT_FORMAT_MASK){
case SND_SOC_DAIFMT_I2S:
    / * 1 - bit data delay * /
```

```
        regs -> rcr2    |= RDATDLY(1);
        regs -> xcr2    |= XDATDLY(1);
        break;
    case SND_SOC_DAIFMT_LEFT_J:
        /* 0 - bit data delay */
        regs -> rcr2    |= RDATDLY(0);
        regs -> xcr2    |= XDATDLY(0);
        regs -> spcr1   |= RJUST(2);
        /* Invert FS polarity configuration */
        //fs 信号极性反
        temp_fmt ^= SND_SOC_DAIFMT_NB_IF;
        break;
    case SND_SOC_DAIFMT_DSP_A:
        /* 1 - bit data delay */
        regs -> rcr2    |= RDATDLY(1);
        regs -> xcr2    |= XDATDLY(1);
        /* Invert FS polarity configuration */
        //fs 信号极性反
        temp_fmt ^= SND_SOC_DAIFMT_NB_IF;
        break;
    case SND_SOC_DAIFMT_DSP_B:
        /* 0 - bit data delay */
        regs -> rcr2    |= RDATDLY(0);
        regs -> xcr2    |= XDATDLY(0);
        /* Invert FS polarity configuration */
        //fs 信号极性反
        temp_fmt ^= SND_SOC_DAIFMT_NB_IF;
        break;
    default:
        /* Unsupported data format */
        return -EINVAL;
    }

    //做 master/slave 的设置
    switch(fmt & SND_SOC_DAIFMT_MASTER_MASK) {
    case SND_SOC_DAIFMT_CBS_CFS:
        /* McBSP master. Set FS and bit clocks as outputs */
        //做 master,fs 和 clk 信号输出
        regs -> pcr0    |= FSXM | FSRM |
                   CLKXM | CLKRM;
        /* Sample rate generator drives the FS */
```

```
        //产生采样信号
        regs -> srgr2        |= FSGM;
        break;
    case SND_SOC_DAIFMT_CBM_CFM:
        /* McBSP slave */
        break;
    default:
        /* Unsupported master/slave configuration */
        return -EINVAL;
    }

    /* Set bit clock(CLKX/CLKR)and FS polarities */
    //设置 bclk 信号和 fs 信号极性
    switch(temp_fmt & SND_SOC_DAIFMT_INV_MASK){
    case SND_SOC_DAIFMT_NB_NF:
        /*
         * Normal BCLK + FS.
         * FS active low. TX data driven on falling edge of bit clock
         * and RX data sampled on rising edge of bit clock.
         */
        //寄存器 fs 和 clk 极性设置都设为 nb + nf
        regs -> pcr0     |= FSXP | FSRP |
                 CLKXP | CLKRP;
        break;
    case SND_SOC_DAIFMT_NB_IF:
        //clk 极性设置设为 nb
        regs -> pcr0     |= CLKXP | CLKRP;
        break;
    case SND_SOC_DAIFMT_IB_NF:
        //fs 极性设置设为 nf
        regs -> pcr0     |= FSXP | FSRP;
        break;
    case SND_SOC_DAIFMT_IB_IF:
        //都不设为 ib + if
        break;
    default:
        return -EINVAL;
    }

    return 0;
}
```

由以上代码分析可见，主要是设计寄存器的操作，在注释中已经对实际的操作进行了说明，具体的寄存器信息在芯片手册中有更详尽的描述。

4. codec 模块

对 codec 模块部分，之前在 codec 的注册函数 snd_soc_register_codec 定义可见其会将 codec driver 和 codec 侧的 dai 同时注册，因为这两者都是与 codec 相关的。这里以 DM3730 板上的 twl4030 为例，介绍相关的框架。首先是 dai 接口说明。

```
static struct snd_soc_dai_driver twl4030_dai[ ] = {
{
	.name = "twl4030 - hifi",
	.playback = {
		.stream_name = "HiFi Playback",
		.channels_min = 2,
		.channels_max = 4,
		.rates = TWL4030_RATES | SNDRV_PCM_RATE_96000,
		.formats = TWL4030_FORMATS,},
	.capture = {
		.stream_name = "Capture",
		.channels_min = 2,
		.channels_max = 4,
		.rates = TWL4030_RATES,
		.formats = TWL4030_FORMATS,},
	.ops = &twl4030_dai_hifi_ops,
},
};
```

可见同处理器侧的 dai 相同，都是进行接口能力的说明和操作接口的说明。操作接口与处理器侧的 dai 功能上是一致的，只是实际的操作要通过处理器的 I^2C 总线操作来完成。

codec 驱动的细节如下：

```
static struct snd_soc_codec_driver soc_codec_dev_twl4030 = {
	.probe = twl4030_soc_probe,
	.remove = twl4030_soc_remove,
	.suspend = twl4030_soc_suspend,
	.resume = twl4030_soc_resume,
	.read = twl4030_read_reg_cache,
	.write = twl4030_write,
	.set_bias_level = twl4030_set_bias_level,
	.reg_cache_size = sizeof(twl4030_reg),
	.reg_word_size = sizeof(u8),
	.reg_cache_default = twl4030_reg,
};
```

这里主要是对 codec 管理的接口，下面来看一下 probe 的实现细节：

```
static int twl4030_soc_probe(struct snd_soc_codec * codec)
{
    struct twl4030_priv * twl4030;

    twl4030 = kzalloc(sizeof(struct twl4030_priv),GFP_KERNEL);
    if(twl4030 == NULL){
        printk("Can not allocate memroy\n");
        return -ENOMEM;
    }
    //设置设备特殊属性
    snd_soc_codec_set_drvdata(codec,twl4030);
    /* Set the defaults,and power up the codec */
    twl4030 -> sysclk = twl4030_codec_get_mclk()/ 1000;
    codec -> idle_bias_off = 1;

    twl4030_init_chip(codec);

    //增加音频控制接口
    snd_soc_add_controls(codec,twl4030_snd_controls,
                ARRAY_SIZE(twl4030_snd_controls));
    //增加 DAPM 接口
    twl4030_add_widgets(codec);
    return 0;
}
```

可见主要是对设备特殊的属性进行设置，另外就是增加控制接口，这样系统就可以通过这些控制接口直接操作 codec。对于 codec 主要是由 codec driver 来提供控制接口，而由 codec dai 来提供数据传输能力。

这样对 DM3730 ALSA SoC 中各个模块的实现都进行了分析和说明。

6.3.5 音频驱动电源管理相关说明

对于音频驱动的电源管理部分，ALSA SoC 框架层提供了基本的电源管理框架，之前在其与底层 platform 关联的 platform driver 中可见有电源管理的操作接口 soc_pm_ops。详细内容如下：

```
static const struct dev_pm_ops soc_pm_ops = {
    .suspend = soc_suspend,
    .resume = soc_resume,
    .poweroff = soc_poweroff,
};
```

主要的电源管理操作就是通过这些接口实现的。下面分析 soc_suspend 的细节：

```
static int soc_suspend(struct device * dev)
{
    struct platform_device * pdev = to_platform_device(dev);
    struct snd_soc_card * card = platform_get_drvdata(pdev);
    int i;

    /* If the initialization of this soc device failed,there is no codec
     * associated with it.  Just bail out in this case.
     */
    if(list_empty(&card -> codec_dev_list))
        return 0;

    //保证 card 处于 power on 状态才能执行 suspend 操作
    snd_power_lock(card -> snd_card);
    snd_power_wait(card -> snd_card,SNDRV_CTL_POWER_D0);
    snd_power_unlock(card -> snd_card);

    //应用层的操作都需要 card 处于 SNDRV_CTL_POWER_D0,这里设置
    //不同的 power 状态保证屏蔽应用层的操作
    snd_power_change_state(card -> snd_card,SNDRV_CTL_POWER_D3hot);

    /* mute any active DAC s */
    //对所有 stream 中的 codec dai 进行 mute 操作
    for(i = 0;i < card -> num_rtd;i ++ ){
        struct snd_soc_dai * dai = card -> rtd[i]. codec_dai;
        struct snd_soc_dai_driver * drv = dai -> driver;

        if(card -> rtd[i]. dai_link -> ignore_suspend)
            continue;

        if(drv -> ops -> digital_mute && dai -> playback_active)
            drv -> ops -> digital_mute(dai,1);
    }

    /* suspend all pcms */
    //对所有的 pcm 进行 stream 操作
    for(i = 0;i < card -> num_rtd;i ++ ){
        if(card -> rtd[i]. dai_link -> ignore_suspend)
            continue;
        //ALSA 框架提供了 pcm 的操作接口,其中会触发具体设备的 trigger
```

```
        //进行相应的 suspend 操作
        snd_pcm_suspend_all(card->rtd[i].pcm);
    }

    if(card->suspend_pre)
        card->suspend_pre(pdev,PMSG_SUSPEND);

    //对处理器侧的 dai 和 platform 进行 suspend 操作,通常相应的操作为空
    for(i = 0;i < card->num_rtd;i ++ ){
        struct snd_soc_dai * cpu_dai = card->rtd[i].cpu_dai;
        struct snd_soc_platform * platform = card->rtd[i].platform;

        if(card->rtd[i].dai_link->ignore_suspend)
            continue;

        if(cpu_dai->driver->suspend && !cpu_dai->driver->ac97_control)
            cpu_dai->driver->suspend(cpu_dai);
        if(platform->driver->suspend && !platform->suspended){
            platform->driver->suspend(cpu_dai);
            platform->suspended = 1;
        }
    }

    /* close any waiting streams and save state */
    //这里进行后续的操作,主要是通过 work 必要时发送 DAPM 的 stop 事件
    for(i = 0;i < card->num_rtd;i ++ ){
        run_delayed_work(&card->rtd[i].delayed_work);
        card->rtd[i].codec->suspend_bias_level = card->rtd[i].codec->bias_level;
    }

    //对各个 stream 发送 DAPM 的 suspend 事件
    for(i = 0;i < card->num_rtd;i ++ ){
        struct snd_soc_dai_driver * driver = card->rtd[i].codec_dai->driver;

        if(card->rtd[i].dai_link->ignore_suspend)
            continue;

        if(driver->playback.stream_name != NULL)
            snd_soc_dapm_stream_event(&card->rtd[i],driver->playback.stream_name,
                SND_SOC_DAPM_STREAM_SUSPEND);

        if(driver->capture.stream_name != NULL)
```

```
            snd_soc_dapm_stream_event(&card->rtd[i],driver->capture.stream_name,
                SND_SOC_DAPM_STREAM_SUSPEND);
    }

    /* suspend all CODECs */
    //
    for(i=0;i<card->num_rtd;i++){
        struct snd_soc_codec *codec = card->rtd[i].codec;
        /* If there are paths active then the CODEC will be held with
         * bias _ON and should not be suspended. */
        if(!codec->suspended && codec->driver->suspend){
            //根据 codec 的偏置电压等级进行相应的设置
            switch(codec->bias_level){
            case SND_SOC_BIAS_STANDBY:
            case SND_SOC_BIAS_OFF:
                codec->driver->suspend(codec,PMSG_SUSPEND);
                codec->suspended = 1;
                break;
            default:
                dev_dbg(codec->dev,"CODEC is on over suspend\n");
                break;
            }
        }
    }

    for(i=0;i<card->num_rtd;i++){
        struct snd_soc_dai *cpu_dai = card->rtd[i].cpu_dai;

        if(card->rtd[i].dai_link->ignore_suspend)
            continue;

        //只有 ac97 才进行该操作
        if(cpu_dai->driver->suspend && cpu_dai->driver->ac97_control)
            cpu_dai->driver->suspend(cpu_dai);
    }

    //soc card 后续的操作
    if(card->suspend_post)
        card->suspend_post(pdev,PMSG_SUSPEND);

    return 0;
}
```

从分析中可见，ALSA 框架提供的电源管理能力主要由 snd_pcm_suspend_all 实现，另外在 ALSA SoC 框架中由各个模块实现相应操作，还有部分功能由 DAPM 实现。

DAPM 是 dynamic audio power management（动态音频电源管理）的缩写。DAPM 的目的是降低音频设备的功耗。DAPM 对应用程序来说是透明的，所有与电源相关的开关都在 ALSA SoC 架构中完成，DAPM 根据当前激活的音频流（playback 或 capture）和 card 中的 mixer 等的配置来决定音频控件模块的打开或关闭。

由于音频设备中会有不同的功能节点，通过这些节点的设置会有完全不同的音频通道，DAPM 本身则需要能够维护这些不同节点的功能，以及相应产生的音频通道，在相关的电源管理操作时则会根据音频通道中各个音频节点进行对应的操作，系统提供的相应接口是 dapm_power_widgets，对应具体的音频节点则是由 snd_soc_dapm_widget 结构进行管理。

总体来说 DAPM 主要是对整个音频流（主要是 audio codec 内部的）中的各个节点以及路径进行管理，这样保证在运行过程中可以根据整个音频流的不同状态进行合理地控制，以减少功耗和保证良好的音频效果。运行时管理的重点是在节点和路径上。

先来看看 DAPM 定义的节点管理实体 snd_soc_dapm_widget，分析如下：

```
struct snd_soc_dapm_widget{
    //类型
    enum snd_soc_dapm_type id;
    char * name;                          /* widget name */
    char * sname; /* stream name */
    //所属的 audio codec
    struct snd_soc_codec * codec;
    //widget 的链表
    struct list_head list;

    /* dapm control */
    //控制寄存器
    short reg;                            /* negative reg = no direct dapm */
    //控制偏移位
    unsigned char shift;                  /* bits to shift */
    //保存的值
    unsigned int saved_value;             /* widget saved value */
    //当前状态的值
    unsigned int value;                   /* widget current value */
    //控制屏蔽值,保证只对该 widget 进行控制
    unsigned int mask;                    /* non - shifted mask */
    //当电源开启的值
    unsigned int on_val;                  /* on state value */
    //当电源关闭的值
    unsigned int off_val;                 /* off state value */
```

```
    //以下是运行时的状态信息
    //power 的状态
    unsigned char power:1;                  /* block power status */
    //power 位是否需要逻辑翻转
    unsigned char invert:1;                 /* invert the power bit */
    //表示 widget 是否激活
    unsigned char active:1;                 /* active stream on DAC,ADC's */
    //表示 widget 是否在音频通路上
    unsigned char connected:1;              /* connected codec pin */
    //初始化的状态,置位表示已经实例化
    unsigned char new:1;                    /* cnew complete */
    //表示该 widget 是否有外部信号连接
    unsigned char ext:1;                    /* has external widgets */
    //强制状态,用于进行电源管理时不对当前状态进行检查,直接操作
    unsigned char force:1;                  /* force state */
    unsigned char ignore_suspend:1;         /* kept enabled over suspend */

    //检查当前 power 状态的接口,不同类型的 widget 对于 power 的状态检查的依据
    //是不同的,DAPM 为不同类型的 widget 提供了不同的接口函数.
    int(*power_check)(struct snd_soc_dapm_widget *w);

    /* external events */
    //表示该 widget 关注的 DAPM 事件类型
    unsigned short event_flags;             /* flags to specify event types */
    //处理 DAPM 事件的接口
    int(*event)(struct snd_soc_dapm_widget *,struct snd_kcontrol *,int);

    /* kcontrols that relate to this widget */
    //与 widget 相关功能控制相关的控制元素信息
    int num_kcontrols;
    const struct snd_kcontrol_new *kcontrols;

    /* widget input and outputs */
    //所有从 source 到该 widget 的 path 链接
    struct list_head sources;
    //所有从该 widget 去往 sink 的 path 链接
    struct list_head sinks;

    /* used during DAPM updates */
    //如果该 widget 需要进行一定的 DAPM 操作,则会加入到对应的链表中
    struct list_head power_list;
};
```

DAPM 提供了一批宏定义，用于相应的 codec 驱动对 widget 进行定义。驱动会根据特定 widget 的特点（如是否能进行 DAPM 事件处理，是否有控制能力）选择宏进行定义。

另外一个重要的执行管理实体是 path，由 snd_soc_dapm_path 进行管理，细节如下：

```
struct snd_soc_dapm_path{
    char * name;
    char * long_name;

    //path 中数据源 widget
    struct snd_soc_dapm_widget * source;
    //path 中数据目的 widget
    struct snd_soc_dapm_widget * sink;
    //path 中的控制元素
    struct snd_kcontrol * kcontrol;

    //path 在当前音频流中的状态
    u32 connect:1;      /* source and sink widgets are connected */
    //在进行 DAPM 操作时是否已经遍历过该 path,避免重复遍历
    u32 walked:1;/* path has been walked */

    //连接时的操作接口,某些 codec 需要,一般为空
    int( * connected)(struct snd_soc_dapm_widget * source,
                    struct snd_soc_dapm_widget * sink);

    //有相同 source widget 的 path 的链表
    struct list_head list_source;
    //有相同 sink widget 的 path 的链表
    struct list_head list_sink;
    //整个 codec 的所有 path 的链表
    struct list_head list;
};
```

可见 path 是连接两个 widget 节点的实体，其中的 kcontrol 是根据相关的 widget 进行创建的，而必要的情况如 mixer 和 mux 会将 kcontrol 添加到应用层。通过应用层对 kcontrol 操作可以使音频路径发生变化。snd_soc_dapm_path 管理动态的 path 信息，而设备中静态的连接信息由 snd_soc_dapm_route 定义，并和 widget 一起在初始化时产生 snd_soc_dapm_path 信息以及对应用的 kcontrol 控制元素。

整个音频路径中有效的音频路径概括为从 DAC 至 output pin、从 input pin 至 ADC、从 input pin 至 output pin、从 DAC 至 ADC 等这几类。而从系统的角度无论何时当任何音频路径中有状态发生变化时，都需要进行全局的路径有效性检查，将整个音频路径的相关节点及配置都进行正确的操作，这个工作在 DAPM 中最终是由 dapm_power_widgets 来实现的，下面对其进行分析。

```
static int dapm_power_widgets(struct snd_soc_codec * codec,int event)
{
    struct snd_soc_card * card = codec -> card;
    struct snd_soc_dapm_widget * w;
    LIST_HEAD(up_list);
    LIST_HEAD(down_list);
    int ret = 0;
    int power;
    int sys_power = 0;

    //遍历 codec 所有的 widget
    list_for_each_entry(w,&codec -> dapm_widgets,list) {
        switch(w -> id) {
        //如果定义为 DAPM,特殊操作的 widget 则加入到合适的操作队列
        case snd_soc_dapm_pre:
            dapm_seq_insert(w,&down_list,dapm_down_seq);
            break;
        case snd_soc_dapm_post:
            dapm_seq_insert(w,&up_list,dapm_up_seq);
            break;

        default:
            //没有 power check 接口,说明没有实例化
            if(!w -> power_check)
                continue;

            //检查需要的 power 状态
            if(!w -> force)
                power = w -> power_check(w);
            else
                power = 1;
            //有 widget 需要 power,那么整个 codec 系统就需要 power
            if(power)
                sys_power = 1;

            if(w -> power == power)
                continue;

            //widget 当前的 power 状态与希望的不同需要将其加入对应的队列
            //需要关闭加入 down 队列,需要打开加入 up 队列
            if(power)
                dapm_seq_insert(w,&up_list,dapm_up_seq);
```

```
            else
                dapm_seq_insert(w,&down_list,dapm_down_seq);

            //这里设置新的 power 状态
            w->power = power;
            break;
        }
    }

    //那些没有通过 widget 进行电源控制的 codec,由系统整体的电源控制
    if(list_empty(&codec->dapm_widgets)){
        switch(event){
        case SND_SOC_DAPM_STREAM_START:
        case SND_SOC_DAPM_STREAM_RESUME:
            sys_power = 1;
            break;
        case SND_SOC_DAPM_STREAM_STOP:
            sys_power = !!codec->active;
            break;
        case SND_SOC_DAPM_STREAM_SUSPEND:
            sys_power = 0;
            break;
        case SND_SOC_DAPM_STREAM_NOP:
            switch(codec->bias_level){
                case SND_SOC_BIAS_STANDBY:
                case SND_SOC_BIAS_OFF:
                    sys_power = 0;
                    break;
                default:
                    sys_power = 1;
                    break;
            }
            break;
        default:
            break;
        }
    }

    //根据当前需要 codec 系统电源情况及偏置电压状态进行正确的设置
    if(sys_power && codec->bias_level == SND_SOC_BIAS_OFF){
        ret = snd_soc_dapm_set_bias_level(card,codec,SND_SOC_BIAS_STANDBY);
        if(ret != 0)
```

```
            pr_err("Failed to turn on bias: %d\n",ret);
    }

    /* If we're changing to all on or all off then prepare */
    if((sys_power && codec->bias_level == SND_SOC_BIAS_STANDBY) ||
        (!sys_power && codec->bias_level == SND_SOC_BIAS_ON)) {
        ret = snd_soc_dapm_set_bias_level(card,codec,SND_SOC_BIAS_PREPARE);
        if(ret != 0)
            pr_err("Failed to prepare bias: %d\n",ret);
    }

    /* Power down widgets first;try to avoid amplifying pops. */
    //为了避免 pop 音先关掉需要关掉的部分
    dapm_seq_run(codec,&down_list,event,dapm_down_seq);

    /* Now power up. */
    //打开需要打开的部分
    dapm_seq_run(codec,&up_list,event,dapm_up_seq);

    /* If we just powered the last thing off drop to standby bias */
    //后续的系统操作,保证偏置电压的正确操作过程
    if(codec->bias_level == SND_SOC_BIAS_PREPARE && !sys_power) {
        ret = snd_soc_dapm_set_bias_level(card,codec,SND_SOC_BIAS_STANDBY);
        if(ret != 0)
            pr_err("Failed to apply standby bias: %d\n",ret);
    }

    /* If we're in standby and can support bias off then do that */
    if(codec->bias_level == SND_SOC_BIAS_STANDBY &&
        codec->idle_bias_off) {
        ret = snd_soc_dapm_set_bias_level(card,codec,SND_SOC_BIAS_OFF);
        if(ret ! =0)
            pr_err("Failed to turn off bias: %d\n",ret);
    }

    /* If we just powered up then move to active bias */
    if(codec->bias_level == SND_SOC_BIAS_PREPARE && sys_power) {
        ret = snd_soc_dapm_set_bias_level(card,codec,SND_SOC_BIAS_ON);
        if(ret != 0)
            pr_err("Failed to apply active bias: %d\n",ret);
    }
```

```
        pop_dbg(codec -> pop_time,"DAPM sequencing finished,waiting %dms\n",
            codec -> pop_time);
        pop_wait(codec -> pop_time);

        return 0;
    }
```

从分析中可见，该过程会把握整个变化，进行正确的设置，从而保证系统根据状态进行正确的切换，在保证功能的情况下，达到功耗尽量低，从而完成整个动态调整的功能。在实现过程中通过将与音频路径相关的 kcontrol 控制元素包含进 DAPM 管理实体，就可以监控系统设置以及应用设置导致的状态变化，从而进行正确的操作。这样就完整实现了整个音频的电源管理功能。

6.4 视频驱动（V4L2）

6.4.1 视频驱动需求

在各种各样的应用设备中，发展最快、变化最多的就是视频设备。之前在介绍帧缓冲驱动时提到的显示设备分辨率长宽比有不同的标准，其中电视标准设备的输出以及 video 视频通道的输出就与视频设备相关。但是视频设备远不止于此，视频设备的发展是与视觉技术和电视技术发展息息相关的，其中涉及视频采集、视频编码/解码、视频传输、视频显示等各个方面，可以说已经形成了一个完整的链条。其在人们的现实生活中也越来越重要。

现如今视频设备已经不完全局限于视频功能，由于视频设备与多媒体的关系很紧密，有些设备还需要有音频的功能，这样视频设备更像是一个容器，包含各种功能。大部分现代视频设备由多个处理器组成，在物理上还需要一定的连接总线来进行控制，这就进一步增加了系统的复杂程度。

面对如此复杂的设备，对视频驱动来说是个不小的挑战，对驱动框架同样有很高的要求。视频驱动需要能够控制这些不同的设备，应用层也要求能够通过简单的控制接口访问到设备的信息，并进行控制；另外还需要能够方便地进行设备特殊的控制；应用还需要能够了解设备内部的各个模块的拓扑情况，并能对不同的数据通道进行灵活的配置。

以上的需求在设备管理方面与音频设备有一些相同，但是由于视频设备的复杂度要更高，所以相应的驱动也更复杂。

6.4.2 视频驱动框架解析

为了满足以上的需求，视频驱动框架分别在不同的层次进行了设计，以实现不同层面的巨大挑战。

1. 应用层设备访问以及控制

要了解应用层如何访问设备还是先从顶层的文件操作开始，视频框架提供的顶层文件操

作接口是 v4l2_fops，详细内容如下：

```
static const struct file_operations v4l2_fops = {
	.owner = THIS_MODULE,
	.read = v4l2_read,
	.write = v4l2_write,
	.open = v4l2_open,
	.mmap = v4l2_mmap,
	.unlocked_ioctl = v4l2_ioctl,
...
	.release = v4l2_release,
	.poll = v4l2_poll,
	.llseek = no_llseek,
};
```

可见与其他类型的功能设备并没有太大的差别，继续从 open 接口了解框架是如何管理其设备的。

```
static int v4l2_open(struct inode *inode,struct file *filp)
{
	struct video_device *vdev;
#if defined(CONFIG_MEDIA_CONTROLLER)
	struct media_entity *entity = NULL;
#endif
	int ret = 0;

	/* Check if the video device is available */
	mutex_lock(&videodev_lock);
	//找到正确的管理实体
	vdev = video_devdata(filp);
	/* return ENODEV if the video device has already been removed. */
	if(vdev == NULL || !video_is_registered(vdev)) {
		mutex_unlock(&videodev_lock);
		return -ENODEV;
	}
	/* and increase the device refcount */
	video_get(vdev);
	mutex_unlock(&videodev_lock);
#if defined(CONFIG_MEDIA_CONTROLLER)
	if(vdev->v4l2_dev && vdev->v4l2_dev->mdev) {
		entity = media_entity_get(&vdev->entity);
		if(!entity) {
			ret = -EBUSY;
```

```
                video_put(vdev);
                return ret;
            }
        }
#endif
    if(vdev->fops->open) {
        if(vdev->lock && mutex_lock_interruptible(vdev->lock)) {
            ret = -ERESTARTSYS;
            goto err;
        }
        //进行实际设备的 open 操作
        if(video_is_registered(vdev))
            ret = vdev->fops->open(filp);
        else
            ret = -ENODEV;
        if(vdev->lock)
            mutex_unlock(vdev->lock);
    }

err:
    /* decrease the refcount in case of an error */
    if(ret) {
#if defined(CONFIG_MEDIA_CONTROLLER)
        if(vdev->v4l2_dev && vdev->v4l2_dev->mdev)
            media_entity_put(entity);
#endif
        video_put(vdev);
    }
    return ret;
}
```

首先代码中出现了 CONFIG_MEDIA_CONTROLLER 宏，这是可配置单元，与设备内拓扑及连接管理相关，后续会进行详细介绍。从代码分析中可见，函数主要是通过 video_devdata 来获得一个 video_device 结构体的指针，紧接着就调用其中的 open 函数，这个 video_device 就是针对应用层的管理实体，相应的所有应用层的调用都会通过其中转到实际设备的操作中。这个管理实体如此重要，相应的系统就应该提供其注册函数，其中底层的注册接口是 _video_register_device，详细分析如下：

```
int __video_register_device(struct video_device *vdev,int type,int nr,
            int warn_if_nr_in_use,struct module *owner)
{
    int i = 0;
```

```
int ret;
int minor_offset = 0;
int minor_cnt = VIDEO_NUM_DEVICES;
const char * name_base;
void * priv = vdev -> dev.p;

/ * A minor value of -1 marks this video device as never
   having been registered * /
vdev -> minor = -1;

/ * the release callback MUST be present * /
WARN_ON( !vdev -> release);
if( !vdev -> release)
      return -EINVAL;

/ * v4l2_fh support * /
spin_lock_init( &vdev -> fh_lock);
INIT_LIST_HEAD( &vdev -> fh_list);

/ * Part 1: check device type * /
//根据设备类型产生基础的设备名字
switch( type) {
case VFL_TYPE_GRABBER:
      name_base = "video";
      break;
case VFL_TYPE_VBI:
      name_base = "vbi";
      break;
case VFL_TYPE_RADIO:
      name_base = "radio";
      break;
case VFL_TYPE_SUBDEV:
      name_base = "v4l-subdev";
      break;
default:
      printk( KERN_ERR "%s called with unknown type: %d\n",
            __func__, type);
      return -EINVAL;
}

vdev -> vfl_type = type;
vdev -> cdev = NULL;
```

```
        //这里与设备层次相关
        if(vdev->v4l2_dev){
            if(vdev->v4l2_dev->dev)
                vdev->parent = vdev->v4l2_dev->dev;
            //这里和特殊的控制相关,关联 v4l2_dev 中的控制元素
            if(vdev->ctrl_handler == NULL)
                vdev->ctrl_handler = vdev->v4l2_dev->ctrl_handler;
        }

        /* Part 2: find a free minor, device node number and device index. */
...
        /* Pick a device node number */
        mutex_lock(&videodev_lock);
        nr = devnode_find(vdev, nr == -1 ? 0 : nr, minor_cnt);
        if(nr == minor_cnt)
            nr = devnode_find(vdev, 0, minor_cnt);
        if(nr == minor_cnt){
            printk(KERN_ERR "could not get a free device node number\n");
            mutex_unlock(&videodev_lock);
            return -ENFILE;
        }
...
        //这里根据之前找到的子设备号设置,注意子设备号与设备索引不同
        vdev->minor = i + minor_offset;
        vdev->num = nr;
        devnode_set(vdev);

        /* Should not happen since we thought this minor was free */
        WARN_ON(video_device[vdev->minor] != NULL);
        vdev->index = get_index(vdev);
        mutex_unlock(&videodev_lock);

        /* Part 3: Initialize the character device */
        //注册字符设备
        vdev->cdev = cdev_alloc();
        if(vdev->cdev == NULL){
            ret = -ENOMEM;
            goto cleanup;
        }
        vdev->cdev->ops = &v4l2_fops;
        vdev->cdev->owner = owner;
        //可见每个子设备号都会创建字符设备
```

```
        ret = cdev_add(vdev -> cdev,MKDEV(VIDEO_MAJOR,vdev -> minor),1);
        if(ret < 0){
            printk(KERN_ERR "%s: cdev_add failed\n",__func__);
            kfree(vdev -> cdev);
            vdev -> cdev = NULL;
            goto cleanup;
        }

        /* Part 4: register the device with sysfs */
        //这里创建 sysfs 设备模型相关信息,与设备文件相关
        memset(&vdev -> dev,0,sizeof(vdev -> dev));
        /* The memset above cleared the device's device_private,so
            put back the copy we made earlier. */
        vdev -> dev.p = priv;
        vdev -> dev.class = &video_class;
        //设备号
        vdev -> dev.devt = MKDEV(VIDEO_MAJOR,vdev -> minor);
        if(vdev -> parent)
            vdev -> dev.parent = vdev -> parent;
        //设备名与设备类型和索引号相关
        dev_set_name(&vdev -> dev,"%s%d",name_base,vdev -> num);
        ret = device_register(&vdev -> dev);
        if(ret < 0){
            printk(KERN_ERR "%s: device_register failed\n",__func__);
            goto cleanup;
        }
        /* Register the release callback that will be called when the last
          reference to the device goes away. */
        vdev -> dev.release = v4l2_device_release;

        if(nr != -1 && nr != vdev -> num && warn_if_nr_in_use)
            printk(KERN_WARNING "%s: requested %s%d,got %s\n",__func__,
                        name_base,nr,video_device_node_name(vdev));
#if defined(CONFIG_MEDIA_CONTROLLER)
        /* Part 5: Register the entity. */
        //此处与设备连接管理相关
        if(vdev -> v4l2_dev && vdev -> v4l2_dev -> mdev){
            vdev -> entity.type = MEDIA_ENTITY_TYPE_DEVNODE_V4L;
            vdev -> entity.name = vdev -> name;
            vdev -> entity.v4l.major = VIDEO_MAJOR;
            vdev -> entity.v4l.minor = vdev -> minor;
            ret = media_device_register_entity(vdev -> v4l2_dev -> mdev,
```

```
                        &vdev -> entity);
                if(ret < 0)
                        printk(KERN_WARNING
                                "%s: media_device_register_entity failed\n",
                                __func__);
        }
#endif
        /* Part 6: Activate this minor. The char device can now be used. */
        //加入系统进行管理
        set_bit(V4L2_FL_REGISTERED,&vdev -> flags);
        mutex_lock(&videodev_lock);
        video_device[vdev -> minor] = vdev;
        mutex_unlock(&videodev_lock);

        return 0;

cleanup:
        mutex_lock(&videodev_lock);
        if(vdev -> cdev)
                cdev_del(vdev -> cdev);
        devnode_clear(vdev);
        mutex_unlock(&videodev_lock);
        /* Mark this video device as never having been registered. */
        vdev -> minor = -1;
        return ret;
}
```

从代码分析中可见，应用层见到的设备文件后面的标号如 video0 并不是子设备号，而是设备的索引号。在注册中还涉及了 ctrl handler 和 v4l2_fh，都与设备对用户开放的功能相关，后续会进行详细介绍。另外其中还包含了设备连接等信息的初始化。从中可以一定程度上了解视频驱动的框架。不管怎样，对于注册函数，首先应该明确 video_device 的作用是针对应用层接口的管理实体。下面先来看看 video_device 的详细内容：

```
struct video_device
{
#if defined(CONFIG_MEDIA_CONTROLLER)
        struct media_entity entity;
#endif
        /* device ops */
        //设备特殊的对应用层的操作接口
        const struct v4l2_file_operations *fops;
```

```
/* sysfs */
struct device dev;                  /* v4l device */
struct cdev *cdev;                  /* character device */

/* Set either parent or v4l2_dev if your driver uses v4l2_device */
//两种模式会对设备管理有不同的影响,由注册的代码可见,这里设置
//与控制相关
struct device *parent;              /* device parent */
struct v4l2_device *v4l2_dev;       /* v4l2_device parent */

/* Control handler associated with this device node.  May be NULL. */
//控制相关
struct v4l2_ctrl_handler *ctrl_handler;

/* device info */
char name[32];
//设备类型
int vfl_type;
/* minor is set to -1 if the registration failed */
//子设备号
int minor;
//索引号
u16 num;
/* use bitops to set/clear/test flags */
//一些状态标记,主要是设备是否已经注册
unsigned long flags;
/* attribute to differentiate multiple indices on one physical device */
int index;

/* V4L2 file handles */
//设备事件相关,主要对应用层处理使用
spinlock_t                          fh_lock;/* Lock for all v4l2_fhs */
struct list_head                    fh_list;/* List of struct v4l2_fh */

int debug;                          /* Activates debug level */

/* Video standard vars */
v4l2_std_id tvnorms;                /* Supported tv norms */
v4l2_std_id current_norm;           /* Current tvnorm */

/* callbacks */
//释放资源接口,主要用于释放 device 中的设备私有管理实体
```

```
        void( * release)(struct video_device * vdev);

        /* ioctl callbacks */
        //设备特殊的 ioctl 接口
        const struct v4l2_ioctl_ops * ioctl_ops;

        /* serialization lock */
        struct mutex * lock;
    };
```

从结构分析中可见，其中提供了两个操作接口：fops 和 ioctl_ops。fops 是针对应用层的操作接口，其形式与文件操作接口基本相似，细节如下：

```
    struct v4l2_file_operations{
        struct module * owner;
        ssize_t( * read)(struct file * ,char __user * ,size_t,loff_t * );
        ssize_t( * write)(struct file * ,const char __user * ,size_t,loff_t * );
        unsigned int( * poll)(struct file * ,struct poll_table_struct * );
        long( * ioctl)(struct file * ,unsigned int,unsigned long);
        long( * unlocked_ioctl)(struct file * ,unsigned int,unsigned long);
        int( * mmap)(struct file * ,struct vm_area_struct * );
        int( * open)(struct file * );
        int( * release)(struct file * );
    };
```

ioctl_ops 是具体设备的 ioctl 接口。一般来说 ioctl 直接在应用层操作接口提供即可，为什么又定义一个 ioctl_ops 来做相关的工作呢？这是由于视频设备的多样性导致其 IO 控制太多太复杂，视频驱动框架提供了 video_ioctl2 作为标准的处理，可以在 fops 中将 ioctl 设置为 video_ioctl2，这样具体的设备只要定义其自身相关的控制接口函数即可。当然驱动也可以完全重新定义自己的 ioctl 来替代系统提供的标准操作。这样提供了灵活性。

由于视频设备的复杂度高，标准的 ioctl 操作无法覆盖到所有的设备，设备还需要能对一定范围的量进行控制的接口，而且还要提供一定的扩展性来使得应用层能够对设备特殊参数进行控制，另外还需要能从设备中获得特殊的事件信息。视频驱动框架也考虑到这两种需求，并提供了解决方法。

首先，看一下设备特殊控制的解决方法，该功能不仅需要对单一功能的设备进行支持，当设备有多个功能模块的时候，也需要能够支持。之前在分析音频设备的时候看到，应用层只有一个控制接口，这样可以将所有的控制都加入到该接口中。但是对视频设备并不完全适用，主要是因为视频设备的控制流和数据流在对应用层的接口 video_device 中很难完全分离，而且某些设备会有多个数据流入口或者出口，这样对一个视频设备来说，会有多个针对应用层的 video_device 实体以及多个视频文件，但是在设备参数的控制方面最好还是有单一的控制流，这就要求多个功能模块的控制部分能够统一到一起进行管理。

这些控制的统一管理是由结构 v4l2_ctrl_handler 来完成的。下面来看一下细节：

```
struct v4l2_ctrl_handler{
    struct mutex lock;
    //实际控制元素的链表
    struct list_head ctrls;
    //用于控制元素排序的链表
    struct list_head ctrl_refs;
    struct v4l2_ctrl_ref * cached;
    struct v4l2_ctrl_ref * * buckets;
    u16 nr_of_buckets;
    int error;
};
```

而实际的控制元素是由 struct v4l2_ctrl 来表示的，其中包含控制值的类型、范围、属性和操作接口等信息。操作接口由 struct v4l2_ctrl_ops 来表示，定义的操作接口内容如下：

```
struct v4l2_ctrl_ops{
    //获得控制值
    int( * g_volatile_ctrl)(struct v4l2_ctrl * ctrl);
    //测试控制值是否有效
    int( * try_ctrl)(struct v4l2_ctrl * ctrl);
    //设置控制值
    int( * s_ctrl)(struct v4l2_ctrl * ctrl);
};
```

内核通过 ioctl 命令对控制进行操作，相关的命令具体包括：

```
VIDIOC_QUERYCTRL,VIDIOC_QUERYMENU,VIDIOC_G_CTRL,VIDIOC_S_CTRL,VIDIOC_G_
EXT_CTRLS,VIDIOC_TRY_EXT_CTRLS,VIDIOC_S_EXT_CTRLS
```

以上是应用层的操作接口，驱动添加控制元素框架也提供了相应的接口。首先要初始化控制管理实体，通过 v4l2_ctrl_handler_init 来完成。

```
int v4l2_ctrl_handler_init(struct v4l2_ctrl_handler * hdl,unsigned nr_of_controls_hint);
```

控制实体系统提供了几种添加的接口函数，具体如下：

```
struct v4l2_ctrl * v4l2_ctrl_new_std(struct v4l2_ctrl_handler * hdl,
    const struct v4l2_ctrl_ops * ops,u32 id,s32 min,s32 max,u32 step,s32 def);
struct v4l2_ctrl * v4l2_ctrl_new_std_menu(struct v4l2_ctrl_handler * hdl,
    const struct v4l2_ctrl_ops * ops,u32 id,s32 max,s32 mask,s32 def);
struct v4l2_ctrl * v4l2_ctrl_new_custom(struct v4l2_ctrl_handler * hdl,
    const struct v4l2_ctrl_config * cfg,void * priv);
```

系统提供了一些标准的 ID 来表示相应的控制元素，例如：

```
#define V4L2_CID_BRIGHTNESS          (V4L2_CID_BASE + 0)
#define V4L2_CID_CONTRAST            (V4L2_CID_BASE + 1)
#define V4L2_CID_SATURATION          (V4L2_CID_BASE + 2)
#define V4L2_CID_HUE                 (V4L2_CID_BASE + 3)
#define V4L2_CID_AUDIO_VOLUME        (V4L2_CID_BASE + 5)
#define V4L2_CID_AUDIO_BALANCE       (V4L2_CID_BASE + 6)
#define V4L2_CID_AUDIO_BASS          (V4L2_CID_BASE + 7)
#define V4L2_CID_AUDIO_TREBLE        (V4L2_CID_BASE + 8)
```

以上只是很少的一部分，可见控制包含各种与设备相关的部分，包括视频、图像调整、编解码、音频等各个方面，这些都是设备的需要。

框架还提供了对所有控制元素统一设置的接口 v4l2_ctrl_handler_setup 来方便设置操作。下面举一个 DM3730 中 ISP 驱动的例子看控制接口是如何使用的。

```
v4l2_ctrl_handler_init(&prev->ctrls,3);
v4l2_ctrl_new_std(&prev->ctrls,&preview_ctrl_ops,V4L2_CID_BRIGHTNESS,
                  ISPPRV_BRIGHT_LOW,ISPPRV_BRIGHT_HIGH,
                  ISPPRV_BRIGHT_STEP,ISPPRV_BRIGHT_DEF);
v4l2_ctrl_new_std(&prev->ctrls,&preview_ctrl_ops,V4L2_CID_CONTRAST,
                  ISPPRV_CONTRAST_LOW,ISPPRV_CONTRAST_HIGH,
                  ISPPRV_CONTRAST_STEP,ISPPRV_CONTRAST_DEF);
v4l2_ctrl_handler_setup(&prev->ctrls);
sd->ctrl_handler = &prev->ctrls;
```

这里 prev 是 ISP 中的一个子设备模块，最终会通过 v4l2_ctrl_add_handler 将其包含的控制元素加入到 ISP 设备中统一管理，而在针对应用层管理实体 video_device 的注册函数中会将其 v4l2_ctrl_handler 指针指向 v4l2_device 相同的 v4l2_ctrl_handler 实体，这样就完成了统一的管理。

另一部分针对应用层的功能是事件的处理，这部分功能需要看一下 v4l2_fh 结构。

```
struct v4l2_fh{
    struct list_head        list;
    struct video_device *vdev;
    struct v4l2_events        *events;/* events,pending and subscribed */
};
```

v4l2_events 中是所有事件的管理实体，下面来看一下详细的内容：

```
struct v4l2_events{
    //等待事件的等待队列
    wait_queue_head_t wait;
```

```
        //管理注册的关注设备的事件队列
        struct list_head        subscribed;                  /* Subscribed events */
        //下面是管理内核中设备产生事件队列
        struct list_head        free;                        /* Events ready for use */
        struct list_head        available;                   /* Dequeueable event */
        unsigned int            navailable;
        unsigned int            nallocated;                  /* Number of allocated events */
        u32                     sequence;
};
```

其操作方式是应用层注册需要关注哪些事件，而内核会对产生的事件加入队列进行管理，并在应用要获得时将事件的详细信息上报。具体的事件是由 v4l2_event 定义，内容如下：

```
struct v4l2_event{
        __u32                           type;
        union{
            struct v4l2_event_vsync vsync;
            __u8                        data[64];
        }u;
        __u32                           pending;
        __u32                           sequence;
        struct timespec                     timestamp;
        __u32                           reserved[9];
};
```

框架提供了相关的操作接口，具体如下：

```
int v4l2_event_init(struct v4l2_fh *fh);
int v4l2_event_alloc(struct v4l2_fh *fh,unsigned int n);
void v4l2_event_free(struct v4l2_fh *fh);
int v4l2_event_dequeue(struct v4l2_fh *fh,struct v4l2_event *event,int nonblocking);
void v4l2_event_queue(struct video_device *vdev,const struct v4l2_event *ev);
int v4l2_event_pending(struct v4l2_fh *fh);
int v4l2_event_subscribe(struct v4l2_fh *fh,struct v4l2_event_subscription *sub);
int v4l2_event_unsubscribe(struct v4l2_fh *fh,struct v4l2_event_subscription *sub);
```

另外内核提供了标准的 ioctl 命令 VIDIOC_DQEVENT、VIDIOC_SUBSCRIBE_EVENT 和 VIDIOC_UNSUBSCRIBE_EVENT 来完成这方面的功能。

通过以上的分析，v4l2_fh 起到容器的功能，既然是容器应该可以承担更多功能，最新的内核已经包含了 v4l2_ctrl_handler，将控制功能包含进来，这样的实现更合理。v4l2_fh 就完全包含了对应用层的功能。

2. 设备层次管理

前面部分主要涉及视频驱动框架中对应用层接口实体的管理，框架还需要提供实际设备的管理层。对特定的视频设备，框架的管理实体是 v4l2_device。

```
struct v4l2_device{
    //设备模型的关联,通常关联到父设备的 device 如 platform device 中的 device
    struct device * dev;
#if defined(CONFIG_MEDIA_CONTROLLER)
    //对设备内拓扑及连接的管理
    struct media_device * mdev;
#endif
    /* used to keep track of the registered subdevs */
    //子设备的链表
    struct list_head subdevs;
    //驱动使用的锁
    spinlock_t lock;
    /* unique device name,by default the driver name + bus ID */
    char name[V4L2_DEVICE_NAME_SIZE];
    /* notify callback called by some sub - devices. */
    //视频设备内部子设备间的通知操作接口,主要用于红外控制器等外部事件通知
    void( * notify)(struct v4l2_subdev * sd,
                unsigned int notification,void * arg);
    /* The control handler. May be NULL. */
    //设备所有的控制接口,对应用层的控制
    struct v4l2_ctrl_handler * ctrl_handler;
    /* BKL replacement mutex. Temporary solution only. */
    struct mutex ioctl_lock;
};
```

v4l2_device 是对整个视频设备的管理实体，其中可能包含多个子功能模块，而这些子功能模块会作为子设备加入到视频设备中进行管理，从结构中可见视频设备管理实体主要是管理功能，并不包含操作接口，所以这里偏重于逻辑管理，具体功能的操作则由子设备来完成，视频框架中对于子设备进行管理，细节如下：

```
struct v4l2_subdev{
    //设备内部拓扑的节点,其中还会管理连接
    struct media_entity entity;
    //列表连接子设备
    struct list_head list;
    struct module * owner;
    //标记子设备特点
    u32 flags;
```

```
	//标记顶层设备
	struct v4l2_device * v4l2_dev;
	//设备操作接口
	const struct v4l2_subdev_ops * ops;
	/ * The control handler of this subdev. May be NULL. * /
	//控制接口
	struct v4l2_ctrl_handler * ctrl_handler;
	/ * name must be unique * /
	char name[V4L2_SUBDEV_NAME_SIZE];
	/ * can be used to group similar subdevs,value is driver - specific * /
	//用于驱动标记一组子设备
	u32 grp_id;
	/ * pointer to private data * /
	//以下是用于驱动标记自身的管理实体
	void * dev_priv;
	void * host_priv;
	/ * subdev device node * /
	//子设备也可以有对应用层的接口
	struct video_device devnode;
	unsigned int initialized;
	/ * number of events to be allocated on open * /
	//对于应用层事件的数目
	unsigned int nevents;
};
```

对子设备特点，系统提供了以下说明：

```
#define V4L2_SUBDEV_FL_IS_I2C          (1U << 0)
#define V4L2_SUBDEV_FL_IS_SPI          (1U << 1)
#define V4L2_SUBDEV_FL_HAS_DEVNODE     (1U << 2)
#define V4L2_SUBDEV_FL_HAS_EVENTS      (1U << 3)
```

前面两个是总线接口，后面两个是设备针对应用层的特点，分别表示对应用层的接口设备文件和对应用层的事件，这样系统会根据这些设置做相应的操作。子设备为什么也需要设备文件接口呢？这是由于视频设备中很多子功能模块同样需要应用层操作，如图像的统计模块。由于特殊事件的存在，也需要应用层进行相应的处理，设备文件是最直接的方法，在设备文件注册时系统会对这种子设备的文件名做特殊处理，以进行区分。

子设备驱动相关的操作接口如下：

```
struct v4l2_subdev_ops{
	const struct v4l2_subdev_core_ops    * core;
	const struct v4l2_subdev_file_ops * file;
```

```
        const struct v4l2_subdev_tuner_ops      * tuner;
        const struct v4l2_subdev_audio_ops      * audio;
        const struct v4l2_subdev_video_ops      * video;
        const struct v4l2_subdev_vbi_ops        * vbi;
        const struct v4l2_subdev_ir_ops         * ir;
        const struct v4l2_subdev_sensor_ops     * sensor;
        const struct v4l2_subdev_pad_ops * pad;
    };
```

从中可见，视频子设备的操作是多种多样的，设备驱动可以根据自身的特点选择合适的操作接口进行设置。由于调用子设备的接口比较复杂，视频驱动框架提供了简易的宏方便接口调用，具体如下：

```
    #define v4l2_subdev_call(sd,o,f,args...)                    \
        (!(sd)? -ENODEV :(((sd)->ops->o &&(sd)->ops->o->f)?     \
            (sd)->ops->o->f((sd),##args): -ENOIOCTLCMD))
```

具体调用如：v4l2_subdev_call(sd,core,g_chip_ident,&chip);

这些操作接口可以在内核中使用，另外也允许应用层对一些接口使用，具体的实现方法需要先来看看子设备的注册接口 v4l2_device_register_subdev，详细的分析如下：

```
    int v4l2_device_register_subdev(struct v4l2_device * v4l2_dev,
                    struct v4l2_subdev * sd)
    {
    #if defined(CONFIG_MEDIA_CONTROLLER)
        struct media_entity * entity = &sd->entity;
    #endif
        struct video_device * vdev;
        int err;

        /* Check for valid input */
        //检查操作
        if(v4l2_dev == NULL || sd == NULL || !sd->name[0])
            return -EINVAL;

        /* Warn if we apparently re-register a subdev */
        WARN_ON(sd->v4l2_dev != NULL);

        if(!try_module_get(sd->owner))
            return -ENODEV;

        /* This just returns 0 if either of the two args is NULL */
```

```
    //首先将控制元素加入整体设备进行管理
    err = v4l2_ctrl_add_handler(v4l2_dev->ctrl_handler,sd->ctrl_handler);
    if(err)
        return err;
#if defined(CONFIG_MEDIA_CONTROLLER)
    /* Register the entity. */
    if(v4l2_dev->mdev){
        //注册进拓扑及连接管理中
        err = media_device_register_entity(v4l2_dev->mdev,entity);
        if(err < 0){
            module_put(sd->owner);
            return err;
        }
    }
#endif
    //加入到整体设备 v4l2_dev 进行设备管理
    sd->v4l2_dev = v4l2_dev;
    spin_lock(&v4l2_dev->lock);
    list_add_tail(&sd->list,&v4l2_dev->subdevs);
    spin_unlock(&v4l2_dev->lock);

    /* Register the device node. */
    //devnode 就是 video_device
    vdev = &sd->devnode;
    strlcpy(vdev->name,sd->name,sizeof(vdev->name));
    //注意这里使用的是 parent 而不是 v4l2_dev,这样在注册时不会设置 ctrl handler
    vdev->parent = v4l2_dev->dev;
    //视频文件的操作接口
    vdev->fops = &v4l2_subdev_fops;
    vdev->release = video_device_release_empty;
    if(sd->flags & V4L2_SUBDEV_FL_HAS_DEVNODE){
        //只有声明需要创建设备文件才进行注册,并声明是 VFL_TYPE_SUBDEV
        err = __video_register_device(vdev,VFL_TYPE_SUBDEV,-1,1,
                        sd->owner);
        if(err < 0){
            v4l2_device_unregister_subdev(sd);
            return err;
        }
    }
#if defined(CONFIG_MEDIA_CONTROLLER)
    entity->v4l.major = VIDEO_MAJOR;
    entity->v4l.minor = vdev->minor;
```

```
#endif
    return 0;
}
```

从代码中可见，子设备文件操作接口定义为 v4l2_subdev_fops，这就是转换操作的入口，具体细节如下：

```
const struct v4l2_file_operations v4l2_subdev_fops = {
    .owner = THIS_MODULE,
    .open = subdev_open,
    .unlocked_ioctl = subdev_ioctl,
    .release = subdev_close,
    .poll = subdev_poll,
};
```

其只定义了部分操作，ioctl 也只定义了一部分操作，主要的操作在控制流上，所以对子设备的应用层接口应该主要关心控制部分。

对于不同类型的硬件操作接口，如 I^{2}C 和 SPI，视频驱动框架还提供了简便的绑定接口，具体如下：

```
struct v4l2_subdev *v4l2_i2c_new_subdev(struct v4l2_device *v4l2_dev,
        struct i2c_adapter *adapter,const char *client_type,
        u8 addr,const unsigned short *probe_addrs);
struct v4l2_subdev *v4l2_i2c_new_subdev_board(struct v4l2_device *v4l2_dev,
        struct i2c_adapter *adapter,struct i2c_board_info *info,
        const unsigned short *probe_addrs,int enable_devnode);
/* Initialize an v4l2_subdev with data from an i2c_client struct */
void v4l2_i2c_subdev_init(struct v4l2_subdev *sd,struct i2c_client *client,
        const struct v4l2_subdev_ops *ops);
/* Return i2c client address of v4l2_subdev. */
unsigned short v4l2_i2c_subdev_addr(struct v4l2_subdev *sd);

struct v4l2_subdev *v4l2_spi_new_subdev(struct v4l2_device *v4l2_dev,
        struct spi_master *master,struct spi_board_info *info,
        int enable_devnode);
/* Initialize an v4l2_subdev with data from an spi_device struct */
void v4l2_spi_subdev_init(struct v4l2_subdev *sd,struct spi_device *spi,
        const struct v4l2_subdev_ops *ops);
```

整体的针对物理设备的管理层次与组织就通过 v4l2_device 和 v4l2_subdev 这两个实体来实现了。

视频驱动系统无论在设备文件层面还是物理设备管理层面都建立了层次关系。在设备文

件层面子设备对应的设备文件只有控制特殊事件的功能，并不作为数据流的出口或者入口，而通常的视频功能设备文件会是数据流的出口或者入口，另外一个物理设备可以有多个数据流的出入口，但是它们都会关联到相同的物理设备管理实体 v4l2_device。

3. 设备内拓扑及连接管理

以上解决的是应用层接口以及设备管理层次的问题，但是设备内部的拓扑以及连接问题还需要解决，而且要提供用户查询设置的接口，这些功能由 media controller 部分完成。

类似于音频设备，视频设备内部的通道也可以非常复杂，会有节点进行视频流通道切换等操作。这就需要在内部对设备的拓扑以及连接进行定义和管理，在外部还要提供应用层操作的接口。视频驱动框架对这部分的管理由 media_device 来提供，具体细节如下：

```
struct media_device{
    /* dev -> driver_data points to this struct. */
    //设备模型中的层次关系,通常 SoC 设备中指向 platform 设备中的 device
    struct device *dev;
    //对应用层的管理实体
    struct media_devnode devnode;

    char model[32];
    char serial[40];
    char bus_info[32];
    u32 hw_revision;
    u32 driver_version;

    //当前下一个管理 entity 应有的 ID 号
    u32 entity_id;
    //管理的 entity 链表
    struct list_head entities;

    /* Protects the entities list */
    spinlock_t lock;
    /* Serializes graph operations. */
    struct mutex graph_mutex;

    //通知相关实体某些事件的发生,使得模块可以进行相关操作
    int(*link_notify)(struct media_pad *source,
            struct media_pad *sink,u32 flags);
};
```

从结构分析中可见，应用层使用 media_devnode 进行管理，而在内部则通过 entity 链表进行管理。先来看看内部的管理，内部具体的管理实体是 media_entity，详细内容如下：

```
struct media_entity{
    //用于 entity 的链表
```

```
struct list_head list;
struct media_device * parent;           /* Media device this entity belongs to */
//在 media device 中的 ID 号
u32 id;                                 /* Entity ID,unique in the parent media device context */
const char * name;                      /* Entity name */
//类型说明
u32 type;                               /* Entity type(MEDIA_ENTITY_TYPE_* ) */
u32 revision;                           /* Entity revision,driver specific */
unsigned long flags;                    /* Entity flags(MEDIA_ENTITY_FLAG_* ) */
//group ID 用于驱动的操作,可以将不同的设备分组
u32 group_id;                           /* Entity group ID */
//模块的数据端点的数目,包括输入或者输出端点
u16 num_pads;                           /* Number of input and output pads */
//与模块相关的连接数目,包括输入和输出的连接
u16 num_links;                          /* Number of existing links,both enabled and disabled */
u16 num_backlinks;                      /* Number of backlinks */
u16 max_links;                          /* Maximum number of links */
//端点的信息
struct media_pad * pads;                /* Pads array(num_pads elements) */
//连接的信息
struct media_link * links;              /* Links array(max_links elements) */

//该模块的操作接口,主要是建立连接时的设置
const struct media_entity_operations * ops;        /* Entity operations */

//该模块的当前数据流数目
int stream_count;                       /* Stream count for the entity. */
//该模块的引用计数,通常用于电源管理操作
int use_count;                          /* Use count for the entity. */

//说明 entity 具体属于哪个数据通道,由驱动进行管理,通常是在
//操作点(如 video_device)所属的 pipeline
struct media_pipeline * pipe;           /* Pipeline this entity belongs to. */

//这里是具体设备的信息,从中可见 media controller 的扩展性
union{
    /* Node specifications */
    struct{
        u32 major;
        u32 minor;
    } v4l;
    struct{
```

```
            u32 major;
            u32 minor;
        }fb;
        struct{
            u32 card;
            u32 device;
            u32 subdevice;
        }alsa;
        int dvb;

        /* Sub-device specifications */
        /* Nothing needed yet */
    };
};
```

实际使用时 media_entity 要嵌入在管理实体中，如之前所见 video_device 和 v4l2_subdev 中都包含了 media_entity，由于它们都涉及连接与管理。通过 type 来区分 entity 的类型，具体分类如下：

```
#define MEDIA_ENTITY_TYPE_DEVNODE
#define MEDIA_ENTITY_TYPE_DEVNODE_V4L
#define MEDIA_ENTITY_TYPE_DEVNODE_FB
#define MEDIA_ENTITY_TYPE_DEVNODE_ALSA
#define MEDIA_ENTITY_TYPE_DEVNODE_DVB

#define MEDIA_ENTITY_TYPE_V4L2_SUBDEV
#define MEDIA_ENTITY_TYPE_V4L2_SUBDEV_SENSOR
#define MEDIA_ENTITY_TYPE_V4L2_SUBDEV_FLASH
#define MEDIA_ENTITY_TYPE_V4L2_SUBDEV_LENS
```

这里 devnode 表示设备文件管理实体中嵌入的 entity，而 subdev 表示子设备中嵌入的 entity。

另外 entity 还包含端点以及连接的属性，这些属性由下面的结构进行管理：

```
struct media_pad{
    //所属的 entity
    struct media_entity *entity;        /* Entity this pad belongs to */
    //entity 中 pad 的编号
    u16 index;                          /* Pad index in the entity pads array */
    //属性
    unsigned long flags;                /* Pad flags(MEDIA_PAD_FLAG_*) */
};
```

```
struct media_link{
    //连接中的数据源端点
    struct media_pad * source;          /* Source pad */
    //连接中的数据目的端点
    struct media_pad * sink;            /* Sink pad */
    struct media_link * reverse;        /* Link in the reverse direction */
    //属性
    unsigned long flags;                /* Link flags(MEDIA_LINK_FLAG_ * ) */
};
```

这些端点以及连接也有属性，相应的属性如下：

```
#define MEDIA_PAD_FLAG_INPUT            (1 <<0)
#define MEDIA_PAD_FLAG_OUTPUT           (1 <<1)
#define MEDIA_LINK_FLAG_ENABLED         (1 <<0)
#define MEDIA_LINK_FLAG_IMMUTABLE       (1 <<1)
#define MEDIA_LINK_FLAG_DYNAMIC         (1 <<2)
```

框架层还提供一些函数对这些管理实体进行操作，相关的接口及说明如下：

```
//entity 的初始化与销毁操作
int media_entity_init(struct media_entity * entity,u16 num_pads,
            struct media_pad * pads,u16 extra_links);
void media_entity_cleanup(struct media_entity * entity);

//连接创建操作
int media_entity_create_link(struct media_entity * source,u16 source_pad,
            struct media_entity * sink,u16 sink_pad,u32 flags);
int __media_entity_setup_link(struct media_link * link,u32 flags);
int media_entity_setup_link(struct media_link * link,u32 flags);

//连接查找操作
struct media_link * media_entity_find_link(struct media_pad * source,
            struct media_pad * sink);
struct media_pad * media_entity_remote_source(struct media_pad * pad);

//entity 使用计数操作
struct media_entity * media_entity_get(struct media_entity * entity);
void media_entity_put(struct media_entity * entity);

//遍历相关操作
void media_entity_graph_walk_start(struct media_entity_graph * graph,
```

```
                    struct media_entity * entity);
struct media_entity * media_entity_graph_walk_next(struct media_entity_graph * graph);

//数据流相关属性设置
void media_entity_pipeline_start(struct media_entity * entity,struct media_pipeline * pipe);
void media_entity_pipeline_stop(struct media_entity * entity);
//entity 接口函数调用的宏
#define media_entity_call(entity,operation,args...)                  \
        (((entity) -> ops &&(entity) -> ops -> operation)?                  \
           (entity) -> ops -> operation((entity),##args):-ENOIOCTLCMD)
```

这里主要是初始化、设置和遍历，以及流属性设置等操作，实际的设备操作驱动会通过 entity 获得嵌入 entity 实际管理的实体进而执行相关的操作。可见在内部 media controller 的实体主要起到对输入输出端点进行抽象，对连接进行抽象，并将端点和连接进行管理，由于其嵌入到实际的设备管理中，所以可以进行由抽象到具体的转换，从而进行实际的操作。对抽象和具体之间的转换，视频驱动框架也提供了相关的宏，细节如下：

```
#define media_entity_to_v4l2_subdev(ent)container_of(ent,struct v4l2_subdev,entity)
#define media_entity_to_video_device(entity)\
        container_of(entity,struct video_device,entity)
```

接下来看看应用层接口管理是如何实现的。相应的管理实体由 media_devnode 来承担，细节如下：

```
struct media_devnode{
    /* device ops */
    //为将来扩展提供的设备相关的操作接口,目前使用统一提供的接口
    const struct media_file_operations * fops;

    /* sysfs */
    //设备模型实体
    struct device dev;              /* media device */
    //字符设备
    struct cdev cdev;               /* character device */
    struct device * parent;         /* device parent */

    /* device info */
    //子设备号
    int minor;
    //访问属性
    unsigned long flags;        /* Use bitops to access flags */
```

```
    /* callbacks */
    void(*release)(struct media_devnode *mdev);
};
```

其属性还是很简单的，也留了一定的扩展余地。下面看看相应的注册管理函数的实现。

```
int __must_check media_devnode_register(struct media_devnode *mdev)
{
    int minor;
    int ret;

    /* Part 1: Find a free minor number */
    mutex_lock(&media_devnode_lock);
    //首先查找子设备号
    minor = find_next_zero_bit(media_devnode_nums,0,MEDIA_NUM_DEVICES);
    if(minor == MEDIA_NUM_DEVICES){
        mutex_unlock(&media_devnode_lock);
        printk(KERN_ERR "could not get a free minor\n");
        return -ENFILE;
    }

    set_bit(mdev->minor,media_devnode_nums);
    mutex_unlock(&media_devnode_lock);

    //设置子设备号
    mdev->minor = minor;

    /* Part 2: Initialize and register the character device */
    //注册字符设备
    cdev_init(&mdev->cdev,&media_devnode_fops);
    mdev->cdev.owner = mdev->fops->owner;

    //注册字符设备
    ret = cdev_add(&mdev->cdev,MKDEV(MAJOR(media_dev_t),mdev->minor),1);
    if(ret < 0){
        printk(KERN_ERR "%s: cdev_add failed\n",__func__);
        goto error;
    }

    /* Part 3: Register the media device */
    //设置设备模型实体
    mdev->dev.bus = &media_bus_type;
```

```
        mdev->dev.devt = MKDEV(MAJOR(media_dev_t),mdev->minor);
        mdev->dev.release = media_devnode_release;
        if(mdev->parent)
            mdev->dev.parent = mdev->parent;
        //设置设备文件名
        dev_set_name(&mdev->dev,"media%d",mdev->minor);
        //注册到设备模型,依据这些信息产生设备文件
        ret = device_register(&mdev->dev);
        if(ret < 0) {
            printk(KERN_ERR "%s: device_register failed\n",__func__);
            goto error;
        }

        /* Part 4: Activate this minor. The char device can now be used. */
        //设置已经注册
        set_bit(MEDIA_FLAG_REGISTERED,&mdev->flags);

        return 0;

    error:
        cdev_del(&mdev->cdev);
        clear_bit(mdev->minor,media_devnode_nums);
        return ret;
    }
```

可见 media controller 会创建单独的设备文件，并提供统一的文件操作接口 media_devnode_fops 用以与应用层交互信息。框架提供了进一步封装的接口 media_device_register 用于整体的管理实体 media_device 注册。

```
    int __must_check media_device_register(struct media_device *mdev)
    {
        int ret;

        if(WARN_ON(mdev->dev == NULL || mdev->model[0] == 0))
            return -EINVAL;

        mdev->entity_id = 1;
        INIT_LIST_HEAD(&mdev->entities);
        spin_lock_init(&mdev->lock);
        mutex_init(&mdev->graph_mutex);

        /* Register the device node. */
```

```
    //media 设备操作的接口
    mdev -> devnode.fops = &media_device_fops;
    mdev -> devnode.parent = mdev -> dev;
    mdev -> devnode.release = media_device_release;
    //设备节点注册
    ret = media_devnode_register(&mdev -> devnode);
    if(ret < 0)
        return ret;
    //创建属性文件
    ret = device_create_file(&mdev -> devnode.dev,&dev_attr_model);
    if(ret < 0){
        media_devnode_unregister(&mdev -> devnode);
        return ret;
    }

    return 0;
}
```

可见操作也十分简单，主要是注册了实际的操作接口 media_device_fops，该接口支持设备文件操作，具体如下：

```
#define MEDIA_IOC_DEVICE_INFO       _IOWR('M',1,struct media_device_info)
#define MEDIA_IOC_ENUM_ENTITIES     _IOWR('M',2,struct media_entity_desc)
#define MEDIA_IOC_ENUM_LINKS        _IOWR('M',3,struct media_links_enum)
#define MEDIA_IOC_SETUP_LINK        _IOWR('M',4,struct media_link_desc)
```

通过这些操作就可以实现各个端点能力的查询以及连接的建立，实现整个系统灵活的查询与控制。

4. 数据流

数据流是视频设备操作的核心。其中包括对数据操作区域、数据格式等，而对数据的操作方式是通过 buffer queue 来进行的，每次操作单个 buffer。相应的数据流操作是通过 ioctl 命令来完成的，系统对于标准的视频设备文件提供的与数据流相关的操作命令如下：

```
VIDIOC_REQBUFS:分配内存
VIDIOC_QUERYBUF:把 VIDIOC_REQBUFS 中分配的数据缓存转换成物理地址
VIDIOC_S_FMT:设置数据 buffer 的格式
VIDIOC_G_FMT:读取数据 buffer 的格式
VIDIOC_TRY_FMT:验证格式是否支持
VIDIOC_CROPCAP:查询设备侧的数据范围能力
VIDIOC_S_CROP:设置设备侧的数据范围
VIDIOC_G_CROP:读取设备侧的数据范围
VIDIOC_QBUF:把数据放回缓存队列交给设备
```

```
VIDIOC_DQBUF:把数据从缓存中取出交给应用层
VIDIOC_STREAMON:启动流
VIDIOC_STREAMOFF:停止流
```

缓冲 buffer 的类型由枚举定义为以下的类型：

```
enum v4l2_buf_type{
    V4L2_BUF_TYPE_VIDEO_CAPTURE                    =1,
    V4L2_BUF_TYPE_VIDEO_OUTPUT                     =2,
    V4L2_BUF_TYPE_VIDEO_OVERLAY                    =3,
    V4L2_BUF_TYPE_VBI_CAPTURE                      =4,
    V4L2_BUF_TYPE_VBI_OUTPUT                       =5,
    V4L2_BUF_TYPE_SLICED_VBI_CAPTURE               =6,
    V4L2_BUF_TYPE_SLICED_VBI_OUTPUT                =7,
#if 1
    /* Experimental */
    V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY =8,
#endif
    V4L2_BUF_TYPE_PRIVATE                          =0x80,
};
```

缓冲的内存分配方式通过枚举包含以下的类型：

```
enum v4l2_memory{
    //驱动分配空间
    V4L2_MEMORY_MMAP              =1,
    //应用层分配空间
    V4L2_MEMORY_USERPTR           =2,
    V4L2_MEMORY_OVERLAY           =3,
};
```

数据流的具体操作与其类型以及内存分配方式都是相关的，例如对于 capture 缓冲类型来说，crop 是设置图像源的取景区域；而对于 output 缓冲类型，crop 是设置显示图像的区域。需要明确的是 format 是针对内存中 buffer 的数据格式属性，crop 是根据设备侧的能力的进行相关的范围设置，实际应用中的缩放都是依据这些参数进行的，而只有理解这些参数的明确意义才能更好地使用它们。在内存分配方式方面对数据流的操作也是有影响的，内存是由应用分配还是内核驱动进行分配，相应的操作也是不同的，主要的差别是在分配内存以及映射操作等方面。

视频设备视频流中的内存管理是一个复杂的过程，包括内存分配、各种流程以及状态的管理，还有映射处理等问题，再加上视频设备数据格式的复杂多样（如 yuv420sp 要求分离的内存块，而 yuv422interleave 则要求单一的数据块)，也加剧了相应管理的复杂性。

内核为了降低视频设备数据流开发的难度，进行了多种技术的开发，从 videobuf 到

videobuf2，以及 CMA、DMA、BUF 等都是为了提升视频设备的性能，并降低视频设备的开发难度。由于很多技术还比较新，在驱动中还没有广泛的应用，所以这里并不进行完整的详述，只是对 videobuf2 以及相关技术进行介绍。

videobuf2 开发的目的是为驱动提供一组用于管理 streaming IO buffer 的接口。videobuf2 实现了三种 buffer 内存管理模型，分别如下：

① vmalloc buffer。这类 buffer 由 vmalloc( )分配，在内核空间虚拟地址上是连续的。

② contiguous DMA buffers。在物理内存上连续，通常硬件设备需要在连续物理地址空间上执行 DMA 操作。

③ S/G DMA buffers。在物理内存上是不连续的，如果硬件上支持 scatter/gather DMA，驱动可以使用 DMA 进行操作。

具体的驱动可以选择这三种的一种方式进行管理，由于每种管理模型都提供了类型为 vb2_mem_ops 的操作接口，具体内容如下：

```
struct vb2_mem_ops{
    //分配 video memory
    void          *( *alloc)(void *alloc_ctx,unsigned long size);
    //通知 buffer 不被使用了
    void          ( *put)(void *buf_priv);
    //当用户分配 memory 类型时,获得一块用户空间 memory 给硬件使用
    void          *( *get_userptr)(void *alloc_ctx,unsigned long vaddr,
                          unsigned long size,int write);
    //通知用户不使用相应空间
    void          ( *put_userptr)(void *buf_priv);
    //返回 buffer 的内核地址空间中的虚拟地址
    void          *( *vaddr)(void *buf_priv);
    //返回给定 buffer 相关的特定私有数据
    void          *( *cookie)(void *buf_priv);
    //返回使用者数目
    unsigned int    ( *num_users)(void *buf_priv);
    //建立 buffer 的应用层映射
    int           ( *mmap)(void *buf_priv,struct vm_area_struct *vma);
};
```

该操作接口定义了内存分配与管理的标准接口，通过为 queue 填入不同的操作接口就可以选择不同的管理模型。

解决了整体上的内存管理方式，相应的 buffer 在不同状态，设备可能需要有不同的操作，为此框架为驱动提供对于 buffer 在不同状态进行设备设置的接口，具体如下：

```
struct vb2_ops{
    // VIDIOC_REQBUFS 和 VIDIOC_CREATE_BUFS 时调用
    int( *queue_setup)(struct vb2_queue *q,const struct v4l2_format *fmt,
            unsigned int *num_buffers,unsigned int *num_planes,
```

```
            unsigned int sizes[ ],void * alloc_ctxs[ ]);
    //锁的操作接口,主要是保证 buffer queue 操作的一致性
    //在 ioctl 需要等待一个新的 buffer 之前调用,这里释放可能会用到的锁
    void( * wait_prepare)(struct vb2_queue * q);
    //在等待结束开始操作时通常是唤醒后调用,重新获得相关的锁
    void( * wait_finish)(struct vb2_queue * q);

    //buffer 的操作接口
    //当由驱动分配时,在分配时调用;当由应用层分配时,在获得一个 buffer 时调用
    //驱动可以进行附加的初始化
    int( * buf_init)(struct vb2_buffer * vb);
    //每次 buffer 从应用层加入 videobuf2 队列时调用,驱动可以执行相关操作
    int( * buf_prepare)(struct vb2_buffer * vb);
    //每次 buffer 从 videobuf2 队列 dequeue 到应用层时调用,驱动可以执行相关操作
    int( * buf_finish)(struct vb2_buffer * vb);
    //与 buf init 对应的释放时调用,驱动可以进行附加的操作
    void( * buf_cleanup)(struct vb2_buffer * vb);

    //stream 状态变化操作接口
    //queue 转入 streaming 状态时的接口,硬件有对 buffer 数目的需求时,
    //驱动可能需要在进入 streaming 时进行相关操作
    int( * start_streaming)(struct vb2_queue * q,unsigned int count);
    //停止 stream,驱动应将所有的 buffer 返回给 videobuf2
    int( * stop_streaming)(struct vb2_queue * q);

    //驱动的 buffer 队列接口,从 videobuf2 队列中获得 buffer,使用该 buffer 进行
    //实际操作
    void( * buf_queue)(struct vb2_buffer * vb);
};
```

以上的接口分别是 videobuf2 框架需要驱动实现的接口（根据设计某些接口可以不实现），框架层更多的是对 buffer 的操作以及管理提供统一的操作接口，还需要将 buffer 转入驱动中进行实际的操作，框架通过驱动提供的 buf_queue 接口将统一管理的队列转入驱动进行操作，相应的需要将 buffer 从驱动转入框架统一管理，框架提供了接口函数 vb2_buffer_done 来进行该操作，驱动也需要将 buffer 的状态信息报告给框架，该接口在驱动操作完一个 buffer 后使用。

以上是驱动的接口，另外 videobuf2 框架为了驱动的应用层操作方便提供了标准的操作函数接口，具体如下：

```
int vb2_querybuf(struct vb2_queue * q,struct v4l2_buffer * b);
int vb2_reqbufs(struct vb2_queue * q,struct v4l2_requestbuffers * req);
int vb2_qbuf(struct vb2_queue * q,struct v4l2_buffer * b);
```

```
int vb2_dqbuf(struct vb2_queue * q,struct v4l2_buffer * b,bool nonblocking);
int vb2_streamon(struct vb2_queue * q,enum v4l2_buf_type type);
int vb2_streamoff(struct vb2_queue * q,enum v4l2_buf_type type);
int vb2_mmap(struct vb2_queue * q,struct vm_area_struct * vma);
unsigned int vb2_poll(struct vb2_queue * q,struct file * file,poll_table * wait);
size_t vb2_read(struct vb2_queue * q,char __user * data,size_t count,
                                        loff_t * ppos,int nonblock);
size_t vb2_write(struct vb2_queue * q,char __user * data,size_t count,
                                        loff_t * ppos,int nonblock);
```

驱动可以直接在 ioctl 中调用相应的接口，从而简化操作并且避免错误。

下面是使用 videobuf2 的例子，主要是在初始化时的操作。

```
q -> type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
q -> io_modes = VB2_MMAP;
q -> drv_priv = handle;
q -> ops = &iss_video_vb2ops;
q -> mem_ops = &vb2_dma_contig_memops;
q -> buf_struct_size = sizeof(struct iss_buffer);
ret = vb2_queue_init(q);
```

可见，这类统一的 buffer queue 管理，主要的功能是在应用层到驱动层之间对 buffer queue 以及 buffer 的操作标准化，在应用层到驱动之间建立一层完整的、健壮的数据管理层，从而简化驱动的开发，避免错误。相应的在接口设计方面，通过对 buffer 不同状态的细化为驱动的操作保留相应的接口，从而提高框架的适用范围，提升整个系统的可移植性以及稳定性。

无论怎样，不同的框架单个 buffer 的管理实体通常是 v4l2_buffer 或者相应的变种，相应的细节如下：

```
struct v4l2_buffer{
    //buffer 的索引号
    __u32                   index;
    //buffer 的类型
    enum v4l2_buf_type          type;
    __u32                   bytesused;
    __u32                   flags;
    enum v4l2_field             field;
    struct timeval          timestamp;
    struct v4l2_timecode        timecode;
    __u32                   sequence;
    //内存的分配方式
    enum v4l2_memory            memory;
```

```
        union{
            //内核分配表示偏移值
            __u32               offset;
            //用户分配方式表示用户地址
            unsigned long       userptr;
        }m;
        __u32                   length;
        __u32                   input;
        __u32                   reserved;
    };
```

相应的信息可以在内核使用，也可以在应用层使用，是一种标准的结构。

关于视频驱动的数据流，还有一部分视频子设备相关的操作。具体操作如下：

```
#define VIDIOC_SUBDEV_G_FMT      _IOWR('V', 4,struct v4l2_subdev_format)
#define VIDIOC_SUBDEV_S_FMT      _IOWR('V', 5,struct v4l2_subdev_format)
#define VIDIOC_SUBDEV_G_FRAME_INTERVAL \
            _IOWR('V',21,struct v4l2_subdev_frame_interval)
#define VIDIOC_SUBDEV_S_FRAME_INTERVAL \
            _IOWR('V',22,struct v4l2_subdev_frame_interval)
#define VIDIOC_SUBDEV_ENUM_MBUS_CODE \
            _IOWR('V', 2,struct v4l2_subdev_mbus_code_enum)
#define VIDIOC_SUBDEV_ENUM_FRAME_SIZE \
            _IOWR('V',74,struct v4l2_subdev_frame_size_enum)
#define VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL \
            _IOWR('V',75,struct v4l2_subdev_frame_interval_enum)
#define VIDIOC_SUBDEV_G_CROP     _IOWR('V',59,struct v4l2_subdev_crop)
#define VIDIOC_SUBDEV_S_CROP     _IOWR('V',60,struct v4l2_subdev_crop)
```

可见只有数据格式的操作，并没有对流的操作。这是由于子设备通常是数据流中的一个节点，而视频设备的数据出口或者入口是对应于内存中的 buffer 的，在单纯的子设备中并没有该功能，也就不需要进行相关的操作了。

从整体上讲，视频设备的数据流管理的主要工作就是保证各个节点格式以及能力的正确设置，并且保证数据的正确操作和处理。

6.4.3 视频驱动应用层操作及框架适配

对视频应用来说，主要就是获取视频流的操作。下面以操作顺序来了解应用层是如何使用视频设备的。该实例是采集设备到显示设备的回显例子，这里简化了代码，只列出采集设备相关的操作。

首先是打开视频设备：

```
int open_cam_device(int flag,int device)
{
    char devnode[20];

    sprintf(devnode,"/dev/video%d",device);
    return open(devnode,flag);
}
```

接下来是对数据格式的控制：

```
int cam_ioctl(int fd,char * pixFormat,char * size,char * sizeH)
{
    struct v4l2_format format;

    int ret = 0;
    int index = 0;

    /* get the current format of the video capture */
    format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ret = ioctl(fd,VIDIOC_G_FMT,&format);
    if(ret < 0){
        perror("VIDIOC_G_FMT");
        return ret;
    }
    //根据参数进行分别率设置
    if(!strcmp(size,"QQCIF")){
        format.fmt.pix.width = 88;
        format.fmt.pix.height = 72;
    }else if(!strcmp(size,"SQCIF")){
        format.fmt.pix.width = 128;
        format.fmt.pix.height = 96;
    }else if
...
    //根据传入的参数进行格式设置
    if(!strcmp(pixFormat,"YUYV"))
        format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
    else if(!strcmp(pixFormat,"UYVY"))
        format.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
    else if(!strcmp(pixFormat,"RGB565"))
        format.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB565;
    else if(!strcmp(pixFormat,"RGB555"))
...
```

```
        /* set size & format of the video image */
        //设置格式
        ret = ioctl(fd,VIDIOC_S_FMT,&format);
        if(ret < 0){
            perror("VIDIOC_S_FMT");
            return ret;
        }

        /* read back */
        //回读一下,可以不需要
        format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        ret = ioctl(fd,VIDIOC_G_FMT,&format);
        if(ret < 0){
            perror("VIDIOC_G_FMT");
            return ret;
        }

        return 0;
    }
```

设置帧率等信息:

```
    int setFramerate(int fd,int framerate)
    {
        struct v4l2_streamparm parm;
        int ret;

        parm.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        ret = ioctl(fd,VIDIOC_G_PARM,&parm);
        if(ret){
            perror("VIDIOC_G_PARM");
            return ret;
        }

        parm.parm.capture.timeperframe.numerator = 1;
        parm.parm.capture.timeperframe.denominator = framerate;
        ret = ioctl(fd,VIDIOC_S_PARM,&parm);
        if(ret){
            perror("VIDIOC_S_PARM");
            return ret;
        }
```

```
        return 0;
    }
```

检查设备是否可以 streaming：

```
    if(ioctl(cfd,VIDIOC_QUERYCAP,&capability) <0){
        perror("cam VIDIOC_QUERYCAP");
        return -1;
    }
    if(capability.capabilities & V4L2_CAP_STREAMING)
        printf("The driver is capable of Streaming!\n");
    else{
        printf("The driver is not capable of Streaming!\n");
        return -1;
    }
```

对 buffer 进行设置：

```
    //查询允许的 buffer 数目
    if(ioctl(cfd,VIDIOC_REQBUFS,&creqbuf) <0){
        perror("cam VIDEO_REQBUFS");
        return -1;
    }

    cbuffers = calloc(creqbuf.count,sizeof( * cbuffers));
    /* mmap driver memory or allocate user memory,and queue each buffer */
    //对每个 buffer 进行设置
    for(i =0;i < creqbuf.count; ++i){
        struct v4l2_buffer buffer;
        buffer.type = creqbuf.type;
        buffer.memory = creqbuf.memory;
        buffer.index = i;
        //获得相应 buffer 的信息
        if(ioctl(cfd,VIDIOC_QUERYBUF,&buffer) <0){
            perror("cam VIDIOC_QUERYBUF");
            return -1;
        }
        if(memtype == V4L2_MEMORY_USERPTR){
            //应用层分配设置参数
            buffer.flags =0;
            //采集使用显示分配的 buffer
            buffer.m.userptr = (unsigned int)vbuffers[i].start;
            buffer.length = vbuffers[i].length;
```

```
        }

        //将 buffer 放入队列进行管理
        if(ioctl(cfd,VIDIOC_QBUF,&buffer) <0){
            perror("cam VIDIOC_QBUF");
            return -1;
        }
    }
```

获得 buffer 并进行相关操作：

```
    //stream on
    if(ioctl(cfd, VIDIOC_STREAMON, &creqbuf.type) <0){
        perror("cam VIDIOC_STREAMON");
        return -1;
    }

    /* caputure 1000 frames or when we hit the passed nmuber of frames */
    cfilledbuffer.type = creqbuf.type;
    i = 0;
    while(i <1000){
        /* De-queue the next avaliable buffer */
        //获得 capture 数据
        while(ioctl(cfd, VIDIOC_DQBUF, &cfilledbuffer) <0)
            perror("cam VIDIOC_DQBUF");
        i++;
        //这里可以进行应用的操作或者相关的设置

        if(i == count){
            //根据设置的 stream off 的数目,停止采集
            printf("Cancelling the streaming capture...\n");
            creqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
            if(ioctl(cfd, VIDIOC_STREAMOFF, &creqbuf.type) <0){
                perror("cam VIDIOC_STREAMOFF");
                return -1;
            }
            printf("Done\n");
            break;
        }

        if(i >= 3){
            //将回显过的 buffer 释放
```

```
            cfilledbuffer.index = vfilledbuffer.index;
            while(ioctl(cfd, VIDIOC_QBUF, &cfilledbuffer) < 0)
                perror("cam VIDIOC_QBUF");
        }
    }
```

最后就是释放的相关操作，基本就是之前操作的逆操作，就不详述了。由代码可见，通过用户分配 buffer 的方式实现输入输出共用相同的 buffer 空间，从而实现无缝连接，并节省内存。视频设备框架的设计是灵活的，可以实现各种复杂的功能。

视频设备在 Android 的适配涉及很多方面，包括 Surface Flinger、HW Composer 以及 Camera 等，这里主要以 Camera 设备的适配进行说明，因为其他部分涉及太多框架的信息。

设备的适配同样需要实现 HAL 层，相应的 HAL 层实体初始化的操作如下：

```
    int camera_device_open(const hw_module_t * module, const char * name,
                    hw_device_t * * device)
    {
        int rv = 0;
        int num_cameras = 1;
        int cameraid;
        V4l2_camera_device_t * camera_device = NULL;
        camera_device_ops_t * camera_ops = NULL;

        if(name! = NULL) {
            cameraid = atoi(name);

            if(cameraid > num_cameras)
            {
                rv = -EINVAL;
                goto fail;
            }

            //android 为 camera 设备定义的抽象的管理实体和操作实体分别是
            //camera_device 和 camera_device_ops_t,这里需要实体化
            camera_device = (V4l2_camera_device_t *)malloc(sizeof( * camera_device));
            if(! camera_device)
            {
                ALOGE("camera_device allocation fail");
                rv = -ENOMEM;
                goto fail;
            }

```

```
camera_ops = (camera_device_ops_t *)malloc(sizeof(*camera_ops));
if(! camera_ops)
{
    ALOGE("camera_ops allocation fail");
    rv = -ENOMEM;
    goto fail;
}

memset(camera_device, 0, sizeof(*camera_device));
memset(camera_ops, 0, sizeof(*camera_ops));

camera_device->base.common.tag = HARDWARE_DEVICE_TAG;
camera_device->base.common.version = 0;
camera_device->base.common.module = (hw_module_t *)(module);
camera_device->base.common.close = camera_device_close;
camera_device->base.ops = camera_ops;

camera_ops->set_preview_window = camera_set_preview_window;
camera_ops->set_callbacks = camera_set_callbacks;
camera_ops->enable_msg_type = camera_enable_msg_type;
camera_ops->disable_msg_type = camera_disable_msg_type;
camera_ops->msg_type_enabled = camera_msg_type_enabled;
camera_ops->start_preview = camera_start_preview;
camera_ops->stop_preview = camera_stop_preview;
camera_ops->preview_enabled = camera_preview_enabled;
camera_ops->store_meta_data_in_buffers = camera_store_meta_data_in_buffers;
camera_ops->start_recording = camera_start_recording;
camera_ops->stop_recording = camera_stop_recording;
camera_ops->recording_enabled = camera_recording_enabled;
camera_ops->release_recording_frame = camera_release_recording_frame;
camera_ops->auto_focus = camera_auto_focus;
camera_ops->cancel_auto_focus = camera_cancel_auto_focus;
camera_ops->take_picture = camera_take_picture;
camera_ops->cancel_picture = camera_cancel_picture;
camera_ops->set_parameters = camera_set_parameters;
camera_ops->get_parameters = camera_get_parameters;
camera_ops->put_parameters = camera_put_parameters;
camera_ops->send_command = camera_send_command;
camera_ops->release = camera_release;
camera_ops->dump = camera_dump;

*device = &camera_device->base.common;
```

```
        // --------TI specific stuff--------
        //实际适配的实体
        camera_device->cameraid = cameraid;
        V4L2CameraHardware = new CameraHardware();
    }
    return rv;
...
}
```

Android 框架加载了相应的 HAL 模块后，就会通过相应的 camera_ops 来操作，而这些操作都是适配操作，实际上是调用 V4L2CameraHardware 中的实际为 CameraHardware 的函数。

接下来以 preview 为例，来看看具体的适配操作的流程：

```
status_t CameraHardware::startPreview()
{
    int width, height;
    int mHeapSize = 0;
    int ret = 0;

    if(! mCamera) {
        delete mCamera;
        //通过封装 V4L2 接口的 camera 的管理层
        mCamera = new V4L2Camera();
    }

    //不同版本内核可能产生的设备文件不同,主要是历史原因,需要适配
    if(version >= KERNEL_VERSION(2,6,37)) {
        if(mCamera->Open(VIDEO_DEVICE_2) < 0)
            return INVALID_OPERATION;
    } else {
        if(mCamera->Open(VIDEO_DEVICE_0) < 0)
            return INVALID_OPERATION;
    }

    Mutex::Autolock lock(mPreviewLock);
    if(mPreviewThread != 0) {
        return INVALID_OPERATION;
    }

    //获得参数
```

```
        mParameters.getPreviewSize(&mPreviewWidth, &mPreviewHeight);
        ALOGD("startPreview width:%d,height:%d",mPreviewWidth,mPreviewHeight);
        if(mPreviewWidth <=0 || mPreviewHeight <=0){
            ALOGE("Preview size is not valid,aborting..Device can not open!!!");
            return INVALID_OPERATION;
        }

        //进行设置
        ret = mCamera->Configure(mPreviewWidth,mPreviewHeight,PIXEL_FORMAT,30);
        if(ret <0){
            ALOGE("Fail to configure camera device");
            return INVALID_OPERATION;
        }
    …
        //进行映射
        ret = mCamera->BufferMap();
        if(ret){
            ALOGE("Camera Init fail: %s", strerror(errno));
            return UNKNOWN_ERROR;
        }

        //开始采集
        ret = mCamera->StartStreaming();
        if(ret){
            ALOGE("Camera StartStreaming fail: %s", strerror(errno));
            mCamera->Uninit();
            mCamera->Close();
            return UNKNOWN_ERROR;
        }

        /* start preview thread */
        //启动 preview 线程进入循环操作
        previewStopped = false;
        mPreviewThread = new PreviewThread(this);

        return NO_ERROR;
    }
```

而在真正的 preview 线程中会通过如下操作将采集的 frame 经过转换进行显示：

```
    if(0 == mapper.lock((buffer_handle_t) *hndl2hndl,CAMHAL_GRALLOC_USAGE, bounds, &dst))
    {
```

```
        // Get preview frame
        //获得采集帧
        tempbuf = mCamera -> GrabPreviewFrame( ) ;

        //进行格式转换以便进行显示
        convertYUYVtoRGB565( ( unsigned char  * )tempbuf,( unsigned char  * )dst, width, height) ;
        mapper. unlock( ( buffer_handle_t) * hndl2hndl) ;
        //加入到显示处理中
        mNativeWindow -> enqueue_buffer( mNativeWindow,( buffer_handle_t * ) hndl2hndl) ;
        //释放采集帧
        mCamera -> ReleasePreviewFrame( ) ;
    }
```

视频设备的实际操作都是通过 V4L2Camera 中的标准 V4L2 接口进行的，例如配置接口。

```
    int V4L2Camera::Configure( int width,int height,int pixelformat,int fps)
    {
        int ret = 0;
        struct v4l2_streamparm parm;

        if( version >= KERNEL_VERSION(2,6,37) )
        {
            videoIn -> width = IMG_WIDTH_VGA;
            videoIn -> height = IMG_HEIGHT_VGA;
            videoIn -> framesizeIn = ( ( IMG_WIDTH_VGA  *  IMG_HEIGHT_VGA) << 1) ;
            videoIn -> formatIn = DEF_PIX_FMT;

            videoIn -> format. fmt. pix. width = IMG_WIDTH_VGA;
            videoIn -> format. fmt. pix. height = IMG_HEIGHT_VGA;
            videoIn -> format. fmt. pix. pixelformat = DEF_PIX_FMT;
        }
    …
        videoIn -> format. type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        do
        {
            ret = ioctl( camHandle, VIDIOC_S_FMT, &videoIn -> format) ;
            if( ret < 0) {
                ALOGE( "Open: VIDIOC_S_FMT Failed: %s", strerror( errno) ) ;
                break;
            }
        } while(0) ;

        return ret;
    }
```

可见就是通过 VIDIOC_S_FMT 实现具体的操作。

Android 的适配在使用流程上与应用的实例是相同的，只是在操作上封装成不同的接口，而且将参数与数据流控制分离，这样方便了管理，也易于与其他模块交互。

6.4.4 TI 芯片视频驱动相关实现详解

对视频驱动的具体实现，以 DM3730 ISP 驱动的设计与实现为例进行介绍。首先来看一下 ISP 硬件框架，如图 6-18 所示。

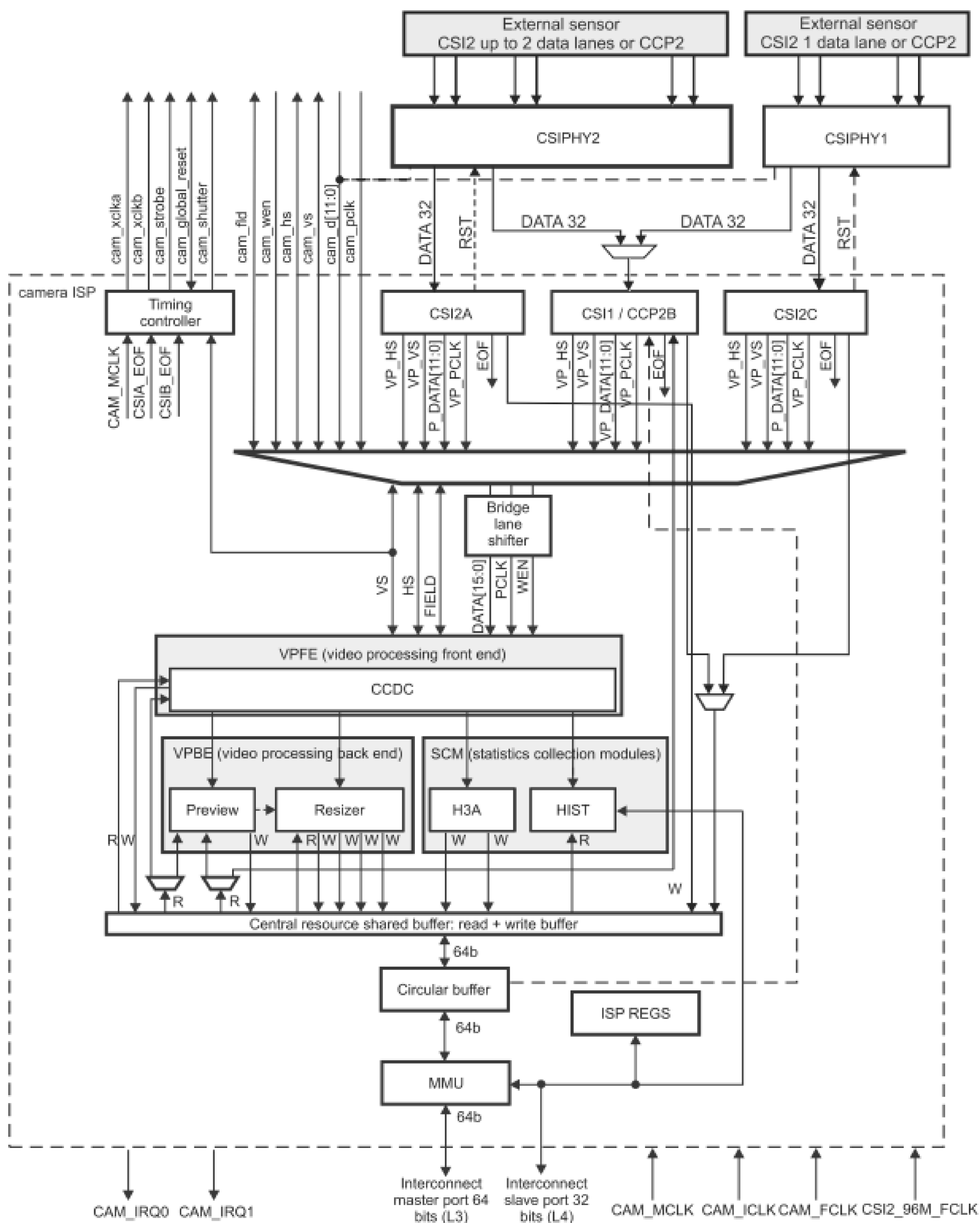


图 6-18　DM3730 ISP 硬件框架

从图 6-18 可见，ISP 模块允许从多个通道读取数据，可以从 CSI、CCP 或者内存，而且硬件模块本身还可以屏蔽，另外其中内嵌 MMU 可以实现帧数据使用物理不连续的内存空间。从硬件看其功能十分强大，这就需要驱动软件支持这些特性，能够容易地设置不同的数据进出以及操作方式，从而更好地使用硬件功能。

1. 整体框架及应用层接口

对这种需要灵活配置的设备，media controller 是最适合的，在实际的驱动设计中也是采用了该方法进行驱动的架构设计。驱动的整体框架如图 6-19 所示。

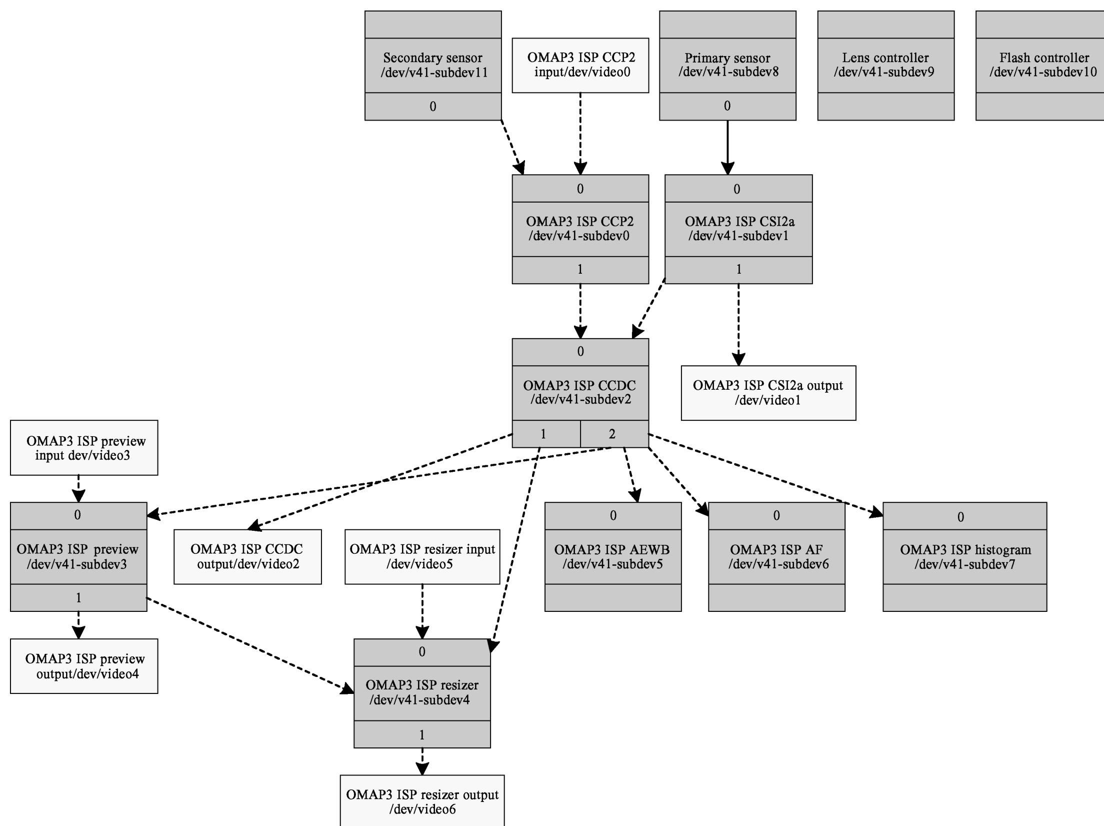


图 6-19　DM3730 ISP 驱动框架

从图 6-19 中可见，其中有很多的节点，每个多矩形组成的节点（如 ISP CCDC）都是子设备是一个功能模块，实现具体的功能，而单一矩形的节点（包含/dev/video?）是 video 设备文件，这些设备文件是数据流的入口或者出口。

ISP 驱动中使用 isp_video 作为以上单一矩形节点的抽象以及管理实体。细节如下：

```
struct isp_video {
    //视频驱动框架的应用层管理实体
    struct video_device video;
    //buffer 类型
    enum v4l2_buf_type type;
```

```
    //相应的 media pad 信息,用于初始化 video 中的 entity 信息
    struct media_pad pad;

    struct mutex mutex;
    atomic_t active;

    //连接到整个 ISP 的管理实体
    struct isp_device *isp;

    //需要的 buffer 空间
    unsigned int capture_mem;
    unsigned int alignment;

    /* Entity video node streaming */
    //streaming 的状态
    unsigned int streaming:1;

    /* Pipeline state */
    //对应的 pipeline,表示具体设备所在的数据 pipeline
    struct isp_pipeline pipe;
    struct mutex stream_lock;

    /* Video buffers queue */
    //针对应用层的 buffer queue
    struct isp_video_queue *queue;
    //针对底层驱动的 buffer queue
    struct list_head dmaqueue;
    enum isp_video_dmaqueue_flags dmaqueue_flags;

    const struct isp_video_operations *ops;
};
```

对有数据输入或者输出的不同的模块会进行相应的注册工作，细节如下：

```
int isp_video_init(struct isp_video *video, const char *name)
{
    const char *direction;
    int ret;

    //根据类型设置 pad 信息
    switch(video->type){
    case V4L2_BUF_TYPE_VIDEO_CAPTURE:
```

```
        direction = "output";
        video -> pad.flags = MEDIA_PAD_FLAG_INPUT;
        break;
case V4L2_BUF_TYPE_VIDEO_OUTPUT:
        direction = "input";
        video -> pad.flags = MEDIA_PAD_FLAG_OUTPUT;
        break;

default:
        return -EINVAL;
}

//初始化 entity 以便进行连接
ret = media_entity_init(&video -> video.entity, 1, &video -> pad, 0);
if(ret < 0)
        return ret;

mutex_init(&video -> mutex);
atomic_set(&video -> active, 0);

spin_lock_init(&video -> pipe.lock);
mutex_init(&video -> stream_lock);

/* Initialize the video device. */
if(video -> ops == NULL)
        video -> ops = &isp_video_dummy_ops;

//设备文件的操作接口
video -> video.fops = &isp_video_fops;
snprintf(video -> video.name, sizeof(video -> video.name),
         "OMAP3 ISP %s %s", name, direction);
//帧类型设备设置
video -> video.vfl_type = VFL_TYPE_GRABBER;
video -> video.release = video_device_release_empty;
//ioctl 控制接口
video -> video.ioctl_ops = &isp_video_ioctl_ops;
//pipeline 状态是 stop
video -> pipe.stream_state = ISP_PIPELINE_STREAM_STOPPED;

//标记驱动的数据以便驱动的应用层接口使用,这里是 isp_video
video_set_drvdata(&video -> video, video);
```

```
        return 0;
    }
```

这里只有初始化，而实际的注册由 isp_video_register 完成，细节如下：

```
    int isp_video_register(struct isp_video *video, struct v4l2_device *vdev)
    {
        int ret;

        video->video.v4l2_dev = vdev;

        //向视频框架注册接口
        ret = video_register_device(&video->video, VFL_TYPE_GRABBER, -1);
        if(ret < 0)
            printk(KERN_ERR "%s: could not register video device(%d)\n",
                    __func__, ret);

        //支持的标准
        video->video.tvnorms            = V4L2_STD_NTSC | V4L2_STD_PAL;
        video->video.current_norm       = V4L2_STD_NTSC;

        return ret;
    }
```

这样注册了实际的视频设备，相应应用层的操作就通过注册的接口进行了。通过一个例子来看看具体的接口是如何实现的。

```
    static int isp_video_try_format(struct file *file, void *fh, struct v4l2_format *format)
    {
        //获得驱动对应的管理实体
        struct isp_video *video = video_drvdata(file);
        struct v4l2_subdev_format fmt;
        struct v4l2_subdev *subdev;
        u32 pad;
        int ret;

        if(format->type != video->type)
            return -EINVAL;

        //查询获得连接的子设备,由 media controller 接口实现
        subdev = isp_video_remote_subdev(video, &pad);
        if(subdev == NULL)
```

```
        return -EINVAL;

    //将像素信息转换成 media bus 信息
    isp_video_pix_to_mbus(&format->fmt.pix, &fmt.format);

    //从 subdev 侧获得信息
    fmt.pad = pad;
    fmt.which = V4L2_SUBDEV_FORMAT_ACTIVE;
    ret = v4l2_subdev_call(subdev, pad, get_fmt, NULL, &fmt);
    if(ret)
        return ret == -ENOIOCTLCMD ? -EINVAL : ret;

    //转换回 pix 信息
    isp_video_mbus_to_pix(video, &fmt.format, &format->fmt.pix);
    return 0;
}
```

可见实际的操作是由子设备完成的，子设备与具体功能相关。

2. 子设备及连接管理

ISP 驱动中每个功能节点都是一个 subdev，而相应的连接也要进行管理，这些都要由具体功能模块来进行，以 CCDC 为例看看是如何实现的。

```
static int isp_ccdc_init_entities(struct isp_ccdc_device *ccdc)
{
    struct v4l2_subdev *sd = &ccdc->subdev;
    struct media_pad *pads = ccdc->pads;
    struct media_entity *me = &sd->entity;
    int ret;

    ccdc->input = CCDC_INPUT_NONE;

    //初始化子设备,主要是将操作初始化,包括各种操作
    v4l2_subdev_init(sd, &ccdc_v4l2_ops);
    strlcpy(sd->name, "OMAP3 ISP CCDC", sizeof(sd->name));
    //驱动定义的组号
    sd->grp_id = 1 << 16;    /* group ID for isp subdevs */
    //设置设备私有数据
    v4l2_set_subdevdata(sd, ccdc);
    //设置有事件上报,有设备文件进行控制操作
    sd->flags |= V4L2_SUBDEV_FL_HAS_EVENTS |
            V4L2_SUBDEV_FL_HAS_DEVNODE;
    //事件缓冲数目,用于框架为应用层缓冲事件
```

```
    sd -> nevents = OMAP3ISP_CCDC_NEVENTS;

    //控制接口初始化
    v4l2_ctrl_handler_init(&ccdc -> ctrls, 1);
    sd -> ctrl_handler = &ccdc -> ctrls;

    //pad 信息初始化
    pads[CCDC_PAD_SINK].flags = MEDIA_PAD_FLAG_INPUT;
    pads[CCDC_PAD_SOURCE_VP].flags = MEDIA_PAD_FLAG_OUTPUT;
    pads[CCDC_PAD_SOURCE_OF].flags = MEDIA_PAD_FLAG_OUTPUT;

    //media controller 操作接口设置
    me -> ops = &ccdc_media_ops;
    //初始化 media entity
    ret = media_entity_init(me, CCDC_PADS_NUM, pads, 0);
    if(ret < 0)
        return ret;

    ccdc_init_formats(sd, NULL);

    //初始化驱动对于应用层的管理实体 isp_video
    ccdc -> video_out.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ccdc -> video_out.ops = &ccdc_video_ops;
    ccdc -> video_out.isp = to_isp_device(ccdc);
    ccdc -> video_out.capture_mem = PAGE_ALIGN(4096 * 4096) * 3;
    ccdc -> video_out.alignment = 32;

    //初始化相应信息
    ret = isp_video_init(&ccdc -> video_out, "CCDC");
    if(ret < 0)
        return ret;

    /* Connect the CCDC subdev to the video node. */
    //将 CCDC 的子设备与应用层设备文件连接起来
    ret = media_entity_create_link(&ccdc -> subdev.entity, CCDC_PAD_SOURCE_OF,
            &ccdc -> video_out.video.entity, 0, 0);
    if(ret < 0)
        return ret;

    return 0;
}
```

同样有注册操作：

```
int isp_ccdc_register_entities(struct isp_ccdc_device *ccdc, struct v4l2_device *vdev)
{
    int ret;

    /* Register the subdev and video node. */
    //注册 subdev
    ret = v4l2_device_register_subdev(vdev, &ccdc->subdev);
    if(ret < 0)
        goto error;

    //注册驱动的应用层管理实体 isp_video
    ret = isp_video_register(&ccdc->video_out, vdev);
    if(ret < 0)
        goto error;

    return 0;

error:
    isp_ccdc_unregister_entities(ccdc);
    return ret;
}
```

这样子设备以及连接就建立起来了。下面来看看连接 media controller 的操作：

```
static int ccdc_link_setup(struct media_entity *entity, const struct media_pad *local,
                const struct media_pad *remote, u32 flags)
{
    struct v4l2_subdev *sd = media_entity_to_v4l2_subdev(entity);
    struct isp_ccdc_device *ccdc = v4l2_get_subdevdata(sd);
    struct isp_device *isp = to_isp_device(ccdc);

    //根据各种输入和输出的情况记录输入的属性
    switch(local->index | media_entity_type(remote->entity)) {
    case CCDC_PAD_SINK | MEDIA_ENTITY_TYPE_V4L2_SUBDEV:
        /* Read from the sensor(parallel interface), CCP2, CSI2a or
         * CSI2c.
         */
        if(!(flags & MEDIA_LINK_FLAG_ENABLED)) {
            ccdc->input = CCDC_INPUT_NONE;
            break;
        }
```

```
		if(ccdc->input != CCDC_INPUT_NONE)
			return -EBUSY;

		if(remote->entity == &isp->isp_ccp2.subdev.entity)
			ccdc->input = CCDC_INPUT_CCP2B;
		else if(remote->entity == &isp->isp_csi2a.subdev.entity)
			ccdc->input = CCDC_INPUT_CSI2A;
		else if(remote->entity == &isp->isp_csi2c.subdev.entity)
			ccdc->input = CCDC_INPUT_CSI2C;
		else
			ccdc->input = CCDC_INPUT_PARALLEL;

		break;

	case CCDC_PAD_SOURCE_VP | MEDIA_ENTITY_TYPE_V4L2_SUBDEV:
		/* Write to preview engine, histogram and H3A. When none of
		 * those links are active, the video port can be disabled.
		 */
		if(flags & MEDIA_LINK_FLAG_ENABLED) {
			if(ccdc->output & ~CCDC_OUTPUT_PREVIEW)
				return -EBUSY;
			ccdc->output |= CCDC_OUTPUT_PREVIEW;
		} else {
			ccdc->output &= ~CCDC_OUTPUT_PREVIEW;
		}
		break;

	case CCDC_PAD_SOURCE_OF | MEDIA_ENTITY_TYPE_DEVNODE:
		/* Write to memory */
		if(flags & MEDIA_LINK_FLAG_ENABLED) {
			if(ccdc->output & ~CCDC_OUTPUT_MEMORY)
				return -EBUSY;
			ccdc->output |= CCDC_OUTPUT_MEMORY;
		} else {
			ccdc->output &= ~CCDC_OUTPUT_MEMORY;
		}
		break;

	case CCDC_PAD_SOURCE_OF | MEDIA_ENTITY_TYPE_V4L2_SUBDEV:
		/* Write to resizer */
		if(flags & MEDIA_LINK_FLAG_ENABLED) {
```

```
                if(ccdc->output & ~CCDC_OUTPUT_RESIZER)
                    return -EBUSY;
                ccdc->output |= CCDC_OUTPUT_RESIZER;
            } else {
                ccdc->output &= ~CCDC_OUTPUT_RESIZER;
            }
            break;

        default:
            return -EINVAL;
        }

        return 0;
    }
```

可见连接建立只是进行输入和输出子设备节点的状态记录，实际操作视频流 streaming on 是在子设备的 s_stream 操作中调用 ccdc_configure 来完成的，而相应的ccdc_configure 则是根据 ccdc_link_setup 时的属性进行相应的寄存器设置。这样就可以完成连接框架到实际设备操作的流程。

3. 视频数据流

关于视频数据流，ISP 驱动框架涉及两个管理实体，其一是 isp_pipeline，表示流的物理 pipeline 路径。细节如下：

```
struct isp_pipeline {
    //media controller 中的 pipeline,在 media controller 实体中指示其所在的 pipeline
    struct media_pipeline pipe;
    spinlock_t lock;
    //pipeline 的状态由枚举 isp_pipeline_state 表示
    unsigned int state;
    //pipeline 中的 stream 的状态
    enum isp_pipeline_stream_state stream_state;
    //pipeline 中的 input 应用层管理实体会对应一个设备文件,如果为空通常是
    //camera sensor 输入
    struct isp_video *input;
    //pipeline 中的 output 应用层管理实体会对应一个设备文件
    struct isp_video *output;
    //时钟相关的设置
    unsigned long l3_ick;
    //pipeline 上各模块的最大时钟
    unsigned int max_rate;
    //采集的帧数
    atomic_t frame_number;
```

```
        bool do_propagation; /* of frame number */
        struct v4l2_fract max_timeperframe;
    };
```

这里主要是各个设备文件所在的数据通道连接状况及其状态，这些会影响到模块的设置。这些状态信息会在电源管理中有相应的功能，比如系统从低功耗状态恢复执行时，需要恢复数据流传送，这时候就会从输入端进行恢复，需要保留这些信息。相应的属性设置在 stream on 操作中执行，相关代码详细内容如下：

```
//获得最远端的设备文件节点,如果为空则表示是 camera sensor
far_end = isp_video_far_end(video);

//根据当前设备文件节点的类型进行 pipeline 属性设置
if(video->type == V4L2_BUF_TYPE_VIDEO_CAPTURE){
    //对于 capture 设备 buffer 是输出到内存中
    state = ISP_PIPELINE_STREAM_OUTPUT | ISP_PIPELINE_IDLE_OUTPUT;
    pipe->input = far_end;
    pipe->output = video;
} else {
    if(far_end == NULL){
        ret = -EPIPE;
        goto error;
    }

    state = ISP_PIPELINE_STREAM_INPUT | ISP_PIPELINE_IDLE_INPUT;
    pipe->input = video;
    pipe->output = far_end;
}
//设置总线吞吐量,主要是会设置总线时钟
omap_pm_set_min_bus_tput(video->isp->dev, OCP_INITIATOR_AGENT, 740000);
pipe->l3_ick = clk_get_rate(video->isp->clock[ISP_CLK_L3_ICK]);

/* Validate the pipeline and update its state. */
//验证 pipeline
ret = isp_video_validate_pipeline(pipe);
if(ret < 0)
    goto error;

//设置 pipeline 状态
spin_lock_irqsave(&pipe->lock, flags);
pipe->state &= ~ISP_PIPELINE_STREAM;
pipe->state |= state;
```

```
    spin_unlock_irqrestore(&pipe->lock, flags);

    if(video->type == V4L2_BUF_TYPE_VIDEO_OUTPUT)
        pipe->max_timeperframe = vfh->timeperframe;

    //对于 buffer queue 的操作
    video->queue = &vfh->queue;
    //初始化驱动的队列
    INIT_LIST_HEAD(&video->dmaqueue);
    atomic_set(&pipe->frame_number, -1);

    //buffer queue 启动
    ret = isp_video_queue_streamon(&vfh->queue);
    if(ret < 0)
        goto error;

    //如果是从 camera sensor 开始,设置硬件进入连续模式
    if(pipe->input == NULL) {
        ret = isp_pipeline_set_stream(pipe,
                        ISP_PIPELINE_STREAM_CONTINUOUS);
        if(ret < 0)
            goto error;
        spin_lock_irqsave(&video->queue->irqlock, flags);
        //此种情况应该驱动 buffer 队列中有 buffer,如果没有 buffer 则是 underrun 异常
        if(list_empty(&video->dmaqueue))
            video->dmaqueue_flags |= ISP_VIDEO_DMAQUEUE_UNDERRUN;
        spin_unlock_irqrestore(&video->queue->irqlock, flags);
    }
```

通过分析可知，很多系统的设置就是通过 pipeline 的状态进行的，其中包括 stream 的操作，对 stream 的操作还要通过 pipeline 的操作完成，相应的接口是 isp_pipeline_enable，详细内容如下：

```
    static int isp_pipeline_enable(struct isp_pipeline *pipe,
                        enum isp_pipeline_stream_state mode)
    {
        struct isp_device *isp = pipe->output->isp;
        struct media_entity *entity;
        struct media_pad *pad;
        struct v4l2_subdev *subdev;
        unsigned long flags;
        int ret = 0;
```

```
spin_lock_irqsave( &pipe -> lock, flags) ;
pipe -> state & = ~ ( ISP_PIPELINE_IDLE_INPUT | ISP_PIPELINE_IDLE_OUTPUT) ;
spin_unlock_irqrestore( &pipe -> lock, flags) ;

pipe -> do_propagation = false;

//先找到输出的实体,然后从输出实体向输入实体进行遍历
entity = &pipe -> output -> video. entity;
while(1) {
    //找到相应的输入 pad,在设置时,如果有输入 pad,就使用 0 号 pad
    //如果 0 号 pad 不是输入 pad,则已经对输入端进行操作,表示遍历结束
    pad = &entity -> pads[0];
    if(! (pad -> flags & MEDIA_PAD_FLAG_INPUT))
        break;

    //向输入端推进
    pad = media_entity_remote_source( pad) ;
    //保证中间是 subdev
    if( pad == NULL || media_entity_type( pad -> entity) ! =
                        MEDIA_ENTITY_TYPE_V4L2_SUBDEV)
        break;

    //通过 entity 获得 subdev 实体
    entity = pad -> entity;
    subdev = media_entity_to_v4l2_subdev( entity) ;

    //对子设备根据 mode 进行 stream 操作
    ret = v4l2_subdev_call( subdev, video, s_stream, mode) ;
    if( ret < 0 && ret! = - ENOIOCTLCMD)
        break;

    if( subdev == &isp -> isp_ccdc. subdev) {
        //如果是 CCDC 需要对 3A 模块进行设置
        v4l2_subdev_call( &isp -> isp_aewb. subdev, video,
                s_stream, mode) ;
        v4l2_subdev_call( &isp -> isp_af. subdev, video,
                s_stream, mode) ;
        v4l2_subdev_call( &isp -> isp_hist. subdev, video,
                s_stream, mode) ;
        pipe -> do_propagation = true;
    }
```

```
        }

        if(pipe->do_propagation && mode == ISP_PIPELINE_STREAM_SINGLESHOT)
            atomic_inc(&pipe->frame_number);

        return ret;
}
```

从中可见，pipeline 就是遍历数据终点到起点的所有子设备，并进行相应的 stream 操作来保证 pipeline 中所有的模块正确工作。

另一个重要的管理实体是 isp_video_queue，这里没有使用 videobuf2 架构是由于在相应驱动开发时内核还没有相关的架构，但是相应的 queue 可以看到 videobuf2 的影子。先来看看的 isp_video_queue 细节：

```
struct isp_video_queue {
    //buffer 的类型
    enum v4l2_buf_type type;
    //驱动提供对应的操作接口
    const struct isp_video_queue_operations *ops;
    struct device *dev;
    unsigned int bufsize;

    unsigned int count;
    //具体的 buffer 管理实体指针
    struct isp_video_buffer *buffers[ISP_VIDEO_MAX_BUFFERS];
    struct mutex lock;
    spinlock_t irqlock;

    unsigned int streaming:1;

    //实际的 buffer 队列
    struct list_head queue;
};
```

这里的 queue 同样是应用层的接口队列，在 isp_video 中可以看到它的指针，而实际驱动操作的 buffer 形成以 dmaqueue 为列表头的列表。实际驱动提供的接口如下：

```
static const struct isp_video_queue_operations isp_video_queue_ops = {
    .queue_prepare = &isp_video_queue_prepare,
    .buffer_prepare = &isp_video_buffer_prepare,
    .buffer_queue = &isp_video_buffer_queue,
```

```
        .buffer_cleanup = &isp_video_buffer_cleanup,
    };
```

其中，isp_video_buffer_queue 的工作就是将 isp_video_queue 中的 buffer 加入到驱动中进行操作，操作结束后，会在中断中根据具体的情况通过 isp_video_buffer_next 将 buffer 归还给 isp_video_queue，以便完成应用的相关操作。层次关系与 videobuf2 是一致的，只是 videobuf2 细化了相关的接口，使用范围也更广。

这样就完成了数据流相关的操作。

4. camera sensor

还剩下一部分就是 camera sensor 的驱动，camera sensor 实际上是通过 I^2C 总线进行连接的外部设备，但是由于与主处理器的 ISP 绑定比较紧密，也为了系统移植的方便，将其做成 subdev 更合适，下面以 mt9t111 sensor 驱动进行说明。

从初始化开始了解：

```
    static int __init mt9t111_init(void)
    {
        return i2c_add_driver(&mt9t111_i2c_driver);
    }
```

I^2C 设备自然需要相应的驱动，对应的匹配是通过 ID table 来完成的，后续介绍总线时会详细介绍。接下来就是具体的 probe：

```
    static int mt9t111_probe(struct i2c_client *client,    const struct i2c_device_id *id)
    {
        struct mt9t111 *mt9t111;
        int ret;

        /* Check if the adapter supports the needed features */
        //检查总线控制器是否支持相应功能
        if(! i2c_check_functionality(client->adapter, I2C_FUNC_SMBUS_BYTE_DATA)) {
            v4l_err(client, "mt9t111: I2C Adapter doesn't support" \
                    " I2C_FUNC_SMBUS_WORD\n");
            return -EIO;
        }

        if(! client->dev.platform_data) {
            v4l_err(client, "No platform data!! \n");
            return -ENODEV;
        }

        //分配管理实体
        mt9t111 = kzalloc(sizeof(*mt9t111), GFP_KERNEL);
```

```
        if(mt9t111 == NULL){
            v4l_err(client, "Could not able to alocate memory!! \n");
            return - ENOMEM;
        }

        //设置设备默认格式等信息
        mt9t111 -> pdata = client -> dev. platform_data;

        mt9t111 -> rect. left = 0;
        mt9t111 -> rect. top = 0;
        mt9t111 -> rect. width = 640;
        mt9t111 -> rect. height = 480;

        mt9t111 -> format. code = V4L2_MBUS_FMT_UYVY8_2X8;
        mt9t111 -> format. width = 640;
        mt9t111 -> format. height = 480;
        mt9t111 -> format. field = V4L2_FIELD_NONE;
        mt9t111 -> format. colorspace = V4L2_COLORSPACE_JPEG;

        //注册子设备有设备文件,没有 event 处理
        v4l2_i2c_subdev_init(&mt9t111 -> subdev, client, &mt9t111_ops);
        mt9t111 -> subdev. flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;

        //只有输出的 pad,只能作为数据源
        mt9t111 -> pad. flags = MEDIA_PAD_FLAG_OUTPUT;
        //创建连接实体
        ret = media_entity_init(&mt9t111 -> subdev. entity, 1, &mt9t111 -> pad, 0);
        if(ret < 0)
            kfree(mt9t111);

        return ret;
    }
```

这样连接实体与子设备的操作实体都建立了，在应用中可以通过如下的代码将 sensor 与 ISP 进行连接：

```
    link. flags |= MEDIA_LINK_FLAG_ENABLED;
    link. source. entity = mediaIn -> mt9t111;
    link. source. index = 0;

    link. source. flags = MEDIA_PAD_FLAG_OUTPUT;
    link. sink. entity = mediaIn -> ccdc;
```

```
        link.sink.index = 0;
        link.sink.flags = MEDIA_PAD_FLAG_INPUT;

        ret = ioctl(mediaIn -> media_fd, MEDIA_IOC_SETUP_LINK, &link);
```

由于后续操作的 sensor 在 pipeline 的连接中，就可以通过 ISP pipeline 的操作来遍历并调用相应子设备的接口，对于 sensor 子设备的接口这里就不进行详述，主要的操作还是通过 I^2C 总线来完成的。

而设备信息的注册则是通过 isp_subdev_i2c_board_info 传入到 ISP 驱动中，并在初始化时，通过 isp_register_subdev_group 来完成。具体如下：

```
static struct v4l2_subdev * isp_register_subdev_group(struct isp_device *isp,
                struct isp_subdev_i2c_board_info *board_info)
{
    struct v4l2_subdev *sensor = NULL;
    unsigned int first;

    if(board_info -> board_info == NULL)
        return NULL;

    //遍历所有的信息
    for(first = 1; board_info -> board_info; ++board_info, first = 0) {
        struct v4l2_subdev *subdev;
        struct i2c_adapter *adapter;

        //检查 I2C 适配器
        adapter = i2c_get_adapter(board_info -> i2c_adapter_id);
        if(adapter == NULL) {
            printk(KERN_ERR "%s: Unable to get I2C adapter %d for "
                "device %s\n", __func__,
                board_info -> i2c_adapter_id,
                board_info -> board_info -> type);
            continue;
        }

        //使用 v4l2 提供的 I2C 辅助接口注册相应的视频子设备,函数内部会注册
        //I2C 设备.
        subdev = v4l2_i2c_new_subdev_board(&isp -> v4l2_dev, adapter,
                board_info -> board_info, NULL, 1);
        if(subdev == NULL) {
            printk(KERN_ERR "%s: Unable to register subdev %s\n",
                __func__, board_info -> board_info -> type);
```

```
                continue;
            }
            if(first)
                sensor = subdev;
        }
        return sensor;
}
```

这里操作的主要目的就是注册 v4l2 的子设备和 I^{2}C 的设备，通过 I^{2}C 总线适配将视频子设备与 I^{2}C 的 Camera Sensor 关联起来，从而实现完整的驱动和设备连接。

6.4.5 视频驱动电源管理相关说明

关于视频驱动的电源管理部分，DM 3730 的 ISP 驱动提供了该部分的功能，主要为低功耗的电源管理，通过 omap3isp_pm_ops 接口提供相关功能，细节如下：

```
static const struct dev_pm_ops omap3isp_pm_ops = {
        .prepare = isp_pm_prepare,
        .suspend = isp_pm_suspend,
        .resume = isp_pm_resume,
        .complete = isp_pm_complete,
};
```

接下来看看具体的实现细节：

```
static int isp_pm_prepare(struct device *dev)
{
        struct isp_device *isp = dev_get_drvdata(dev);
        int reset;

        if(isp->ref_count == 0)
                return 0;
        //suspend 所有的模块
        reset = isp_suspend_modules(isp);
        //关闭中断
        isp_disable_interrupts(isp);
        //包含上下文
        isp_save_ctx(isp);
        if(reset)
                isp_reset(isp);

        return 0;
}
```

```
static int isp_pm_suspend(struct device *dev)
{
    struct isp_device *isp = dev_get_drvdata(dev);

    WARN_ON(mutex_is_locked(&isp->isp_mutex));
    //如果在使用时 suspend 关闭时钟
    if(isp->ref_count)
        isp_disable_clocks(isp);

    return 0;
}

static int isp_pm_resume(struct device *dev)
{
    struct isp_device *isp = dev_get_drvdata(dev);

    if(isp->ref_count == 0)
        return 0;
    //恢复 suspend 时关掉的时钟
    return isp_enable_clocks(isp);
}

static void isp_pm_complete(struct device *dev)
{
    struct isp_device *isp = dev_get_drvdata(dev);

    if(isp->ref_count == 0)
        return;

    //恢复上下文
    isp_restore_ctx(isp);
    //打开中断
    isp_enable_interrupts(isp);
    //恢复所有的模块
    isp_resume_modules(isp);
}
```

在所有的 modules 的 suspend 和 resume 操作过程中，会根据 pipeline 的状态进行 pipeline 相关操作，通过以下函数实现，具体细节如下：

```
static void isp_pipeline_resume(struct isp_pipeline *pipe)
{
```

```
        //检查模式是连续模式还是单帧模式
        int singleshot = pipe -> stream_state == ISP_PIPELINE_STREAM_SINGLESHOT;

        //先恢复输出和输入模块的模式
        isp_video_resume(pipe -> output, ! singleshot);
        if(singleshot)
            isp_video_resume(pipe -> input, 0);
        //恢复 pipeline 的状态
        isp_pipeline_enable(pipe, pipe -> stream_state);
    }

    static void isp_pipeline_suspend(struct isp_pipeline  * pipe)
    {
        //disable pipeline,这样 stream 就停止了
        isp_pipeline_disable(pipe);
    }
```

模块的动态电源管理与 streaming 的状态紧密结合，在子设备的操作中实现。以 CCDC 为例，ccdc_set_stream 完成了相关的功能。

```
    static int ccdc_set_stream(struct v4l2_subdev  * sd, int enable)
    {
        struct isp_ccdc_device  * ccdc = v4l2_get_subdevdata(sd);
        struct isp_device  * isp = to_isp_device(ccdc);
        int ret = 0;

        if(ccdc -> state == ISP_PIPELINE_STREAM_STOPPED){
            if(enable == ISP_PIPELINE_STREAM_STOPPED)
                return 0;
            //当前 CCDC 的状态是关闭,则需要先恢复模块再进行设置
            isp_subclk_enable(isp, OMAP3_ISP_SUBCLK_CCDC);
            isp_reg_set(isp, OMAP3_ISP_IOMEM_CCDC, ISPCCDC_CFG,
                                  ISPCCDC_CFG_VDLC);

            ccdc_configure(ccdc);

            ispccdc_config_vp(ccdc);
            ispccdc_enable_vp(ccdc, 0);
            ccdc -> error = 0;
            ispccdc_print_status(ccdc);
        }
```

```
        switch(enable){
        case ISP_PIPELINE_STREAM_CONTINUOUS:
            if(ccdc->output & CCDC_OUTPUT_MEMORY)
                isp_sbl_enable(isp, OMAP3_ISP_SBL_CCDC_WRITE);

            if(ccdc->underrun || !(ccdc->output & CCDC_OUTPUT_MEMORY))
                ispccdc_enable(ccdc);

            ccdc->underrun = 0;
            break;

        case ISP_PIPELINE_STREAM_SINGLESHOT:
            if(ccdc->output & CCDC_OUTPUT_MEMORY &&
                ccdc->state != ISP_PIPELINE_STREAM_SINGLESHOT)
                isp_sbl_enable(isp, OMAP3_ISP_SBL_CCDC_WRITE);

            ispccdc_enable(ccdc);
            break;

        case ISP_PIPELINE_STREAM_STOPPED:
            //需要关闭 stream 则关闭其中的模块
            ret = ispccdc_disable(ccdc);
            if(ccdc->output & CCDC_OUTPUT_MEMORY)
                isp_sbl_disable(isp, OMAP3_ISP_SBL_CCDC_WRITE);
            isp_subclk_disable(isp, OMAP3_ISP_SUBCLK_CCDC);
            ccdc->underrun = 0;
            break;
        }

        ccdc->state = enable;
        return ret;
    }
```

从中可见，子模块会根据 streaming 的状态进行相应的电源管理操作，这样低功耗与动态电源管理的功能就完整了。

6.5　小结

本章主要介绍各种功能型设备驱动，各种驱动会根据设备的数据特点进行设计，并尽量为应用层提供良好的界面，方便用户应用程序的开发。

功能性驱动各具特色，其中框架设计也包含了很多设计思想。这里主要以层次的方式分析，将整体的框架从针对应用的上层到驱动设备操作的下层进行完整的分析，当从上至下打通时，整个框架就清晰地展现了。

第 7 章　设备驱动之总线型驱动

7.1　内部集成电路总线（I^2C）

7.1.1　I^2C 总线驱动需求

I^2C（Inter – Integrated Circuit）总线是一种由 Philips 公司开发的两线式串行总线（分别是时钟信号 SCL 和数据信号 SDA）。如果带有高速扩展，则最高可到 3.4 MHz。其具有简单高效，总线占用空间小，使用芯片引脚少，互连成本低，可以进行多设备互连等优点，被广泛应用于处理器与外围设备的连接。

由于传感器技术的大力发展，且不需要大量的数据交换，所以 I^2C 总线已被广泛用于处理器连接各种传感器。可以说 I^2C 已经成为嵌入式设备不可或缺的总线形式。

从形式上看 I^2C 是主/从形式总线，在任何时间点上只能有一个主控。嵌入式处理器通常作为 I^2C 总线的主设备，而传感器等外围设备是总线上的从设备。从设备通常有固化的地址，主设备通过地址与从设备通信。

I^2C 总线连接的基本形式如图 7–1 所示。图 7–1 引自《DM 3730 芯片手册》中第 2777 页框图。

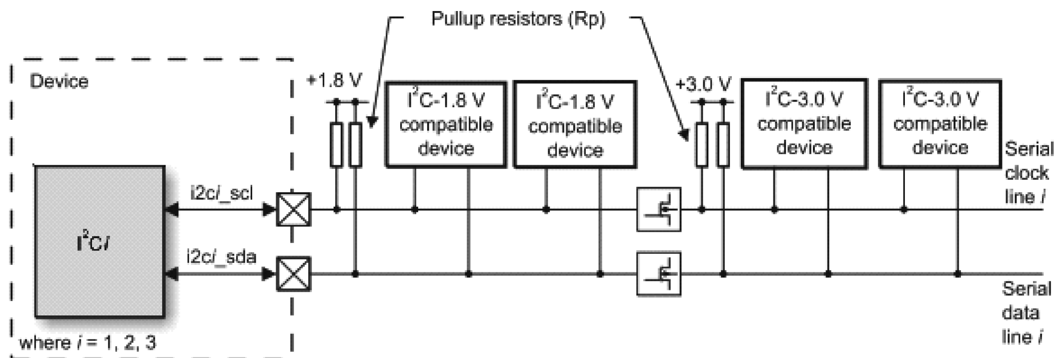


图 7–1　I^2C 总线连接基本形式框图

从图 7–1 可见，处理器中可以有多个 I^2C 总线控制器，每个总线控制器形成一个单独的 I^2C 总线，一条 I^2C 总线允许连接多个设备，而且设备允许使用不同的接口电压，这些不同接口电压的设备分布要合理。如芯片的 IO 接口电压是 1.8 V，所以 1.8 V 的设备要接近芯片，而高电压 3.0 V 接口的设备则通过 level shift 实现升压后进行连接。

总线可以连接各种设备，在物理设备上包含总线控制器以及总线设备。另外系统的硬件连接需要具有灵活性，不同的总线控制器可以连接不同的总线设备，也就是说总线控制器与总线设备要独立，不能够彼此相关。在软件方面也需要具有高度的灵活性，设备信息要与驱

动无关，总线控制器的操作方法要与属性无关，还要在软件层面满足总线控制器与总线设备无关。

另外从设备的角度来看，I²C 总线设备有具体的功能，而总线是实现功能的媒介，通过总线传输命令、状态信息等。这就需要总线系统提供信息交互的接口及规范，即总线数据传输的信号规范。I²C 总线数据传输信号规范如图 7-2 所示。图 7-2 引自《DM 3730 芯片手册》中第 2778 页框图。

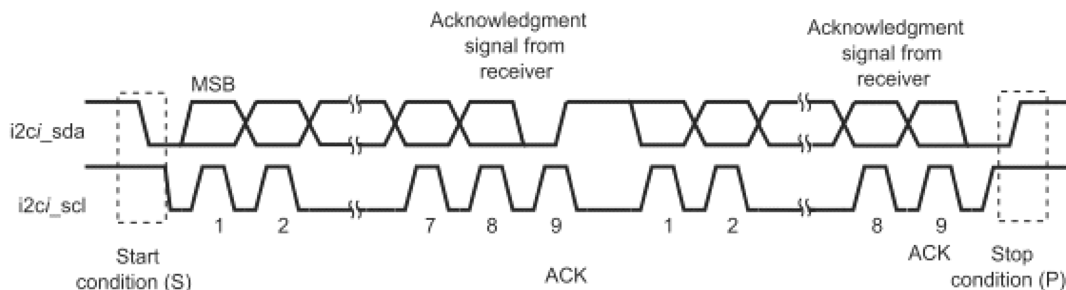


图 7-2　I²C 总线数据传输信号规范

从图 7-2 可见，具体传输是由 Start 和 Stop 标记的，并且每传送一个字长都需要有 ACK 标记。通常传输信号的正确性是由物理的控制器来保证的，也可以通过 GPIO 进行模拟，但是需要软件进行配合。

总体上来说，系统对 I²C 总线驱动的需求就是要能实现总线的各种功能，并且满足以上的各种无关性需求。

7.1.2　I²C 总线驱动框架解析

1. 总线相关以及核心框架

在介绍设备模型的时候，已经介绍了设备模型中的总线抽象管理，通常总线都有自动 probe 功能，即可以在向设备模型注册设备或者驱动的时候，进行绑定操作，由具体总线实例化相关操作。了解总线相关操作对于了解整个总线框架非常有帮助。

在介绍系统初始化的时候，已经看到整个 I²C 的初始化函数 i2c_init 定义为 postcore_initcall，这样就在 I²C 设备或者驱动注册之前进行注册，从而保证总线的功能。而初始化中包含如下语句：

```
retval = bus_register( &i2c_bus_type) ;
```

这里注册了总线类型，其中 bus_register 会设置总线自动 probe 功能。而 i2c_bus_type 可以作为了解整个总线框架的入口，详细定义如下：

```
struct bus_type i2c_bus_type = {
    . name          = "i2c",
    . match         = i2c_device_match,
    . probe         = i2c_device_probe,
```

```
    .remove             = i2c_device_remove,
    .shutdown           = i2c_device_shutdown,
    .pm                 = &i2c_device_pm_ops,
};
```

这里重点的接口是 match 以及匹配之后的 probe。下面详细分析这两个接口：

```
static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
    //检查是否是总线设备
    struct i2c_client       *client = i2c_verify_client(dev);
    struct i2c_driver       *driver;
    //非总线设备直接返回
    if(!client)
        return 0;
...
    driver = to_i2c_driver(drv);
    /* match on an id table if there is one */
    //匹配驱动的 ID 表,表中声明支持的设备,具体的匹配方法是比较字符串
    if(driver->id_table)
        return i2c_match_id(driver->id_table, client) != NULL;

    return 0;
}

static int i2c_device_probe(struct device *dev)
{
    //检查是否是总线设备
    struct i2c_client       *client = i2c_verify_client(dev);
    struct i2c_driver       *driver;
    int status;
    //非总线设备则直接返回
    if(!client)
        return 0;

    driver = to_i2c_driver(dev->driver);
    if(!driver->probe || !driver->id_table)
        return -ENODEV;
    //在 I2C 总线层面将设备与对应的驱动绑定
    client->driver = driver;
    //如果该 I2C 设备可以唤醒系统则进行标记
    if(!device_can_wakeup(&client->dev))
```

```
            device_init_wakeup(&client->dev,
                        client->flags & I2C_CLIENT_WAKE);
        dev_dbg(dev, "probe\n");

        //进行总线设备驱动的探测初始化工作
        status = driver->probe(client, i2c_match_id(driver->id_table, client));
        if(status) {
            client->driver = NULL;
            i2c_set_clientdata(client, NULL);
        }
        return status;
    }
```

从分析中可见，除了基本的操作之外，两个接口函数都通过 i2c_verify_client 进行了设备的验证。下面来看看具体的内容。

```
    struct i2c_client *i2c_verify_client(struct device *dev)
    {
        return(dev->type == &i2c_client_type)
            ? to_i2c_client(dev) : NULL;
    }
```

这里 dev->type 可以是一个特殊的设备类型 i2c_client_type。在设备模型中已经介绍了，device 中的 type 属性在功能类型和总线类型等大的类型中对设备进行细分，并进行相应操作，这里就是实际应用。

进行 type 的划分说明不止一种设备，下面来看看 I^2C 总线的不同设备类型：

```
    struct device_type i2c_client_type;
    struct device_type i2c_adapter_type;
```

其中有两种类型的设备，分别是 i2c_client 类型和 i2c_adapter 类型。进行这样的分类是由于从抽象的角度考虑，总线控制器也是设备，而作为总线的控制接口，其物理上代表的是总线，而不是总线上连接的设备，所以在类型上加以区分。而在进行总线级别的匹配时，需要进行匹配的是总线上连接的设备和其相应的驱动，这就要求确保只有总线设备进行匹配操作，所以才有之前进行检查的代码。

I^2C 总线框架主要就是对这两类设备及操作进行管理，整体的框架如图 7-3 所示。

由图 7-3 可见，I^2C 核心管理的实体就是 i2c_client、i2c_adapter 与总线设备相关的驱动 i2c_driver。为了管理这些实体，提供了接口函数，后面在相关部分会对其进行介绍。另外除了以上的两类设备之外，系统还提供了针对应用层的操作接口管理实体 i2c_dev。下面来看看细节：

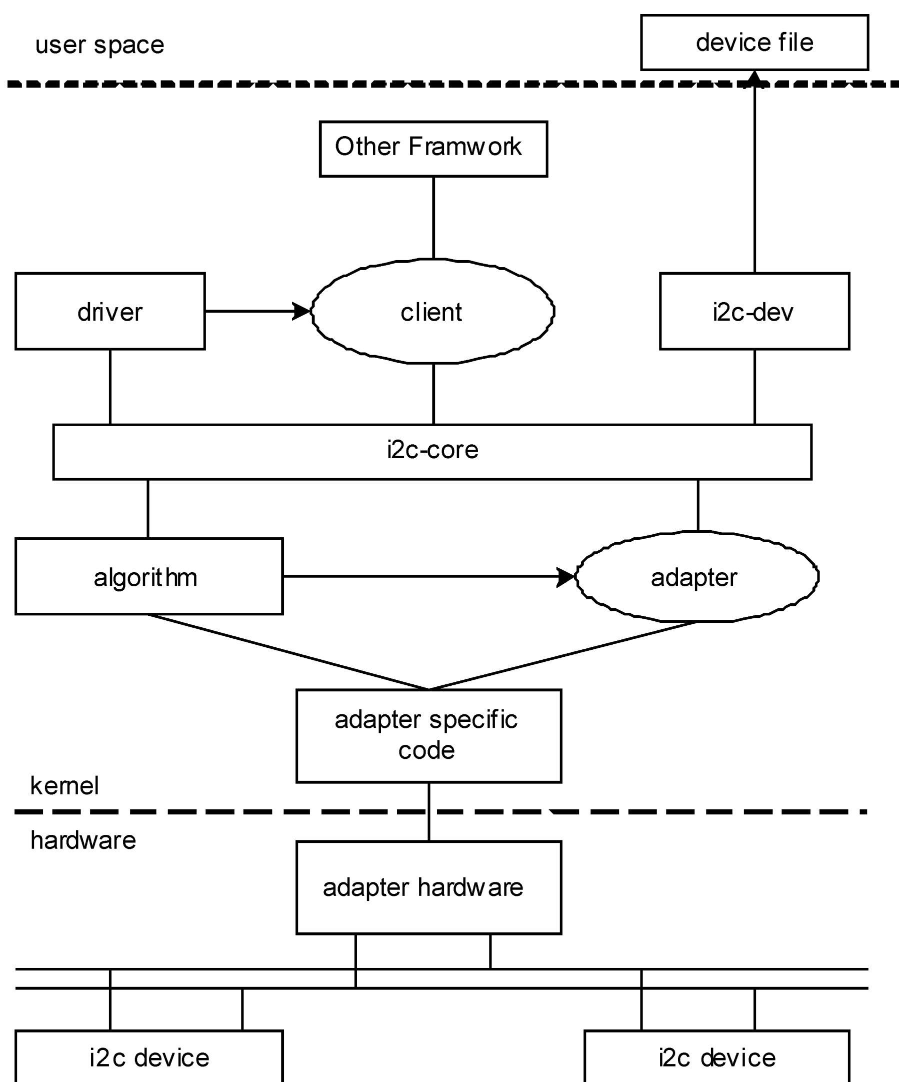


图 7-3　I^2C 总线内核框架

```
struct i2c_dev {
    struct list_head list;
    struct i2c_adapter *adap;
    struct device *dev;
};
```

其中提供了 i2c_adapter 的指针，这是由于对应用层开放的设备文件主要提供 I^2C 应用层驱动的功能，作为驱动主要是对总线设备的操作，通过总线控制器进行总线操作，所以这里只要提供相应的总线控制器即可，而具体与哪个 I^2C 从设备交互及相关的地址等信息都是通过 ioctl 命令设置的。这样对下层的操作都是相同的，只是在应用层接口部分实现 i2c_client 作为操作的接口即可。下面来看看 I^2C 设备文件操作接口 i2cdev_fops 中的 open 函数：

```
static int i2cdev_open(struct inode *inode, struct file *file)
{
    //获得子设备号
    unsigned int minor = iminor(inode);
    struct i2c_client *client;
    struct i2c_adapter *adap;
    struct i2c_dev *i2c_dev;
    //获得管理实体,对应于设备文件
    i2c_dev = i2c_dev_get_by_minor(minor);
    if(!i2c_dev)
        return -ENODEV;
    //获得总线控制器实体
    adap = i2c_get_adapter(i2c_dev->adap->nr);
    if(!adap)
        return -ENODEV;
    //这里创建的 i2c_client 只是个桩,并不会注册到 I²C 核心中,后面可以通过
    //ioctl 命令设置 slave 地址,这样只作为文件操作接口保存必要的信息
    client = kzalloc(sizeof(*client), GFP_KERNEL);
    if(!client){
        i2c_put_adapter(adap);
        return -ENOMEM;
    }
    snprintf(client->name, I2C_NAME_SIZE, "i2c-dev %d", adap->nr);
    client->driver = &i2cdev_driver;

    //绑定总线控制器
    client->adapter = adap;
    //与文件相关联
    file->private_data = client;

    return 0;
}
```

从代码中可见，设置了一个特殊的 driver 即 i2cdev_driver，下面来查看一下细节：

```
static struct i2c_driver i2cdev_driver = {
    .driver = {
        .name       = "dev_driver",
    },
    .attach_adapter     = i2cdev_attach_adapter,
    .detach_adapter     = i2cdev_detach_adapter,
};
```

这里只有 attach_adapter 和 detach_adapter 两个操作接口，这两个接口是系统用于处理新的总线控制器加入或者退出系统时的相关操作，在此情况下驱动可以通过相应接口进行相关操作。应用接口层利用该功能进行相关的操作，通过检测总线控制器的状态变化，进行 i2c_dev 以及设备文件的创建和销毁。

I^2C 设备更多的是对内核的各种功能框架提供设备的控制接口，这样就要求将功能框架设备与 I^2C 设备关联。以视频框架为例，相应的关联是在 v4l2_i2c_subdev_init 中实现的，具体细节如下：

```
void v4l2_i2c_subdev_init(struct v4l2_subdev *sd, struct i2c_client *client,
                const struct v4l2_subdev_ops *ops)
{
    v4l2_subdev_init(sd, ops);
    sd->flags |= V4L2_SUBDEV_FL_IS_I2C;
    /* the owner is the same as the i2c_client's driver owner */
    sd->owner = client->driver->driver.owner;
    /* i2c_client and v4l2_subdev point to one another */
    //将 I2C 设备与 V4L2 子设备关联
    v4l2_set_subdevdata(sd, client);
    i2c_set_clientdata(client, sd);
    /* initialize name */
    snprintf(sd->name, sizeof(sd->name), "%s %d-%04x",
        client->driver->driver.name, i2c_adapter_id(client->adapter),
        client->addr);
}
```

这样 V4L2 子设备就可以在操作中通过 v4l2_get_subdevdata 来获得 I^2C 设备进行相应的操作，而 I^2C 设备移除时也可以通过 i2c_get_clientdata 来获得 V4L2 子设备进行释放操作。

这样 I^2C 框架就分别给内核功能模块与应用层提供了操作接口，在该设计中可以保证共用的部分尽量多。

2. 总线控制器相关

I^2C 总线框架对总线控制器的管理主要是通过 i2c_adapter 来实现的，其详细分析如下：

```
struct i2c_adapter {
    struct module *owner;

    //现内核已经使用 IDR(ID 号与指针对应的机制),该 ID 不建议使用
    unsigned int id __deprecated;
    //总线功能类型,有一般控制类型 HWMON、DDC(Display Data Channel)以及
    // SPD(Serial Presence Detect)等不同类型
    unsigned int class;           /* classes to allow probing for */
    //总线控制器设备特殊的操作总线的方法
```

```
        const struct i2c_algorithm *algo; /* the algorithm to access the bus */
        //具体总线控制器所需的私有数据
        void *algo_data;

        /* data fields that are valid for all devices      */
        //由于总线连接多个设备,该锁保护单一总线操作的原子性
        struct rt_mutex bus_lock;
        //超时值,超时则退出传送
        int timeout;                       /* in jiffies */
        //单一总线操作重传次数
        int retries;
        //设备模型管理
        struct device dev;             /* the adapter device */
        //设备中总线控制器的编号,也可以代表总线编号,可以由具体设备指定
        int nr;
        char name[48];
        struct completion dev_released;

        struct mutex userspace_clients_lock;
        struct list_head userspace_clients;
    };
```

从中可见，除了属性信息外最重要的就是 i2c_algorithm 了。i2c_algorithm 中定义了操作接口，通过这些操作接口可以实现标准的 I^2C 总线操作。由于不同设备的操作方法不同，该接口实际是实现具体设备总线控制器的总线操作逻辑。下面来看看具体细节：

```
    struct i2c_algorithm {
        //标准 I²C 总线传输的操作接口
        int(*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
        //smbus 传输的操作接口
        int(*smbus_xfer)(struct i2c_adapter *adap, u16 addr, unsigned short flags,
                char read_write, u8 command, int size, union i2c_smbus_data *data);
        /* To determine what the adapter supports */
        //检查总线控制器支持的功能,通常由驱动调用来检查总线适配器功能
        //是否满足需求,功能如 I2C_FUNC_I2C 定义在 i2c.h 中
        u32(*functionality)(struct i2c_adapter *);
    };
```

从中可见，有两个操作接口，但是两个并不都需要提供，通常只要提供标准 I^2C 接口 master_xfer 即可。通过标准的 I^2C 操作可以模拟 smbus 操作，I^2C 总线框架提供函数 i2c_smbus_xfer_emulated 来模拟相应的操作。在一次传输中可以批量做多次处理，每个处理都是一次 i2c_msg 的交互操作，具体 i2c_msg 的个数由参数 num 指定。

为了管理总线控制器，I²C 总线框架提供了以下接口：

```
//由系统指定总线控制器编号进行管理
int i2c_add_adapter(struct i2c_adapter *);
//需要按照设备指定的总线控制器编号进行管理
int i2c_add_numbered_adapter(struct i2c_adapter *);
int i2c_del_adapter(struct i2c_adapter *);
```

具体设备的总线控制器在调用以上接口进行管理前，相关的属性（包括 algo）都需要初始化，也就是说框架并没有提供总线控制器的操作 algo 与管理实体 adapter 进行绑定的接口。这是由于 adapter 的很多属性都需要初始化，而与哪个操作算法绑定开发者最清楚，所以直接作为初始化的一部分设置即可。

另外还提供了 adapter 设备特殊管理数据的设置和获取接口，如 i2c_set_adapdata 和 i2c_get_adapdata。

当一个 adapter 加入系统后，说明增加了一个 I²C 总线连接，而通常 I²C 总线是在硬件板上直接进行连接，这就需要进行总线的遍历以及总线设备与相应驱动的绑定。这个流程是在 i2c_register_adapter 中完成的，而该函数会在 i2c_add_adapter 和 i2c_add_numbered_adapter 中进行调用。i2c_register_adapter 中主要工作就是进行 adapter 设备模型的注册，另外通过如下代码进行总线遍历：

```
mutex_lock(&core_lock);
bus_for_each_drv(&i2c_bus_type, NULL, adap, __process_new_adapter);
mutex_unlock(&core_lock);
```

这里对每个 I²C 设备驱动都执行__process_new_adapter，在__process_new_adapter 中会通过 i2c_detect 调用 i2c_detect_address 来由驱动 detect 匹配的设备。如果 detect 成功则注册新设备。其中设备模型和总线会负责绑定驱动，从而完成加载操作，这就实现了总线设备发现相关功能。

总线的主要功能（包括总线传输事务以及总线设备发现）就都在 I²C 总线控制器部分有了完整的支持。

3. 总线设备相关

I²C 总线框架提供了 i2c_client 来完成总线设备管理功能，详细内容如下：

```
struct i2c_client {
    //指示设备的特点如唤醒系统能力,10bit 地址等
    unsigned short flags;          /* div., see below          */
    //地址
    unsigned short addr;           /* chip address - NOTE: 7bit */
    char name[I2C_NAME_SIZE];
    //关联的总线视频器
    struct i2c_adapter *adapter;   /* the adapter we sit on     */
```

```
    //关联的总线驱动
    struct i2c_driver *driver;        /* and our access routines      */
    //设备模型实体
    struct device dev;                /* the device structure         */
    //设备关联到系统的中断号,通常是外部中断信号或者 gpio
    int irq;                          /* irq issued by device         */
    struct list_head detected;
};
```

从相关属性来看，主要的属性是 addr、adapter 和 driver。下面来看看与之相关联的 driver 的详细信息：

```
struct i2c_driver {
    unsigned int class;
    //在总线控制器增加或减少时需要进行的特殊操作
    int(*attach_adapter)(struct i2c_adapter *);
    int(*detach_adapter)(struct i2c_adapter *);

    /* Standard driver model interfaces */
    //标准的设备模型操作,用于设备绑定的初始化以及设备移除的资源释放
    int(*probe)(struct i2c_client *, const struct i2c_device_id *);
    int(*remove)(struct i2c_client *);

    /* driver model interfaces that don' t relate to enumeration    */
    //电源管理相关接口
    void(*shutdown)(struct i2c_client *);
    int(*suspend)(struct i2c_client *, pm_message_t mesg);
    int(*resume)(struct i2c_client *);
    //协议特殊的操作
    void(*alert)(struct i2c_client *, unsigned int data);
    //设备特殊的命令接口
    int(*command)(struct i2c_client *client, unsigned int cmd, void *arg);

    //设备模型接口
    struct device_driver driver;
    //支持设备的 ID 表,主要是 name
    const struct i2c_device_id *id_table;

    /* Device detection callback for automatic device creation */
    //检测是否支持相应设备
    int(*detect)(struct i2c_client *, struct i2c_board_info *);
    const unsigned short *address_list;
```

```
    struct list_head clients;
};
```

从 i2c_driver 的内容分析可见，其主要负责总线相关以及电源管理相关的操作，操作接口中并不涉及实际的信息传输，可以说该驱动主要是处理总线和设备模型相关的事务，并不涉及功能的事务。功能型的事务涉及操作的具体内容，是在功能型驱动中通过 I^2C 的接口实现的。而功能型的事务需要有总线设备的信息，这就要求功能型驱动管理实体中能够包含 i2c_client 的信息。这个桥梁是与 I^2C 总线事务相关的，所以该工作自然就由 i2c_driver 来完成，具体的接口就是 probe，这样整体上各种实体就关联起来了。具体例子见 mt9t111_probe：

```
static int mt9t111_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    struct mt9t111 *mt9t111;
    int ret;

    /* Check if the adapter supports the needed features */
    //检查总线控制器功能是否支持
    if(!i2c_check_functionality(client->adapter, I2C_FUNC_SMBUS_BYTE_DATA)){
        v4l_err(client, "mt9t111: I2C Adapter doesn't support" \
                    " I2C_FUNC_SMBUS_WORD\n");
        return -EIO;
    }
...
    //i2c_client 与 v4l2 子设备关联
    v4l2_i2c_subdev_init(&mt9t111->subdev, client, &mt9t111_ops);
...
    return ret;
}
```

只有总线设备与功能管理实体关联后才能完成功能性的工作，而总线设备对应的驱动主要工作是建立总线设备与功能管理实体的关联；对物理总线以及电源管理等事件响应并进行相应的操作。

以上无论是 i2c_client 还是 i2c_driver 都属于 I^2C 总线框架管理的运行时的实体，而 I^2C 总线设备通常直接焊接在板子上面，这些固定的信息同样需要进行表示，在创建动态管理的设备实体时需要这些信息。在 I^2C 总线框架中相应的实体是 i2c_board_info，内容如下：

```
struct i2c_board_info {
    //设备名
    char        type[I2C_NAME_SIZE];
    /指示设备的特点如唤醒系统能力、10bit 地址等
    unsigned short flags;
```

```
        //地址
        unsigned short addr;
        //特殊信息,会作为 device 中的特殊信息传给 I²C 驱动做处理
        void          * platform_data;
        struct dev_archdata  * archdata;
    ...
        //设备给系统的中断信息
        int           irq;
    };
```

基本的信息就是 type 和 addr，可以通过宏 I2C_BOARD_INFO 来产生。

对 I^2C 总线设备的创建，框架提供的函数接口如下：

```
    //已知设备地址使用的接口
    struct i2c_client * i2c_new_device(struct i2c_adapter * adap,
                        struct i2c_board_info const * info);
    //未知设备地址使用的接口
    struct i2c_client * i2c_new_probed_device(struct i2c_adapter * adap,
                        struct i2c_board_info * info, unsigned short const * addr_list,
                        int( * probe)(struct i2c_adapter * , unsigned short addr))
```

从中可见，创建总线设备需要 i2c_adapter 和 i2c_board_info，对于特定总线设备有 i2c_board_info 信息，这就需要明确与 i2c_adapter 关联。

有两种方法来实现：方法一通过 struct i2c_adapter * i2c_get_adapter(int id)获得指定的 i2c_adapter，然后调用 i2c_new_device 来创建总线设备。方法二是在初始化阶段调用如下函数，注册某个 I^2C 总线上的所有设备信息：i2c_register_board_info(int busnum, struct i2c_board_info const * info, unsigned len)，总线控制器 i2c_adapter 会通过 i2c_scan_static_board_info 来遍历这些信息,为总线上的设备创建 i2c_client。

方法一主要用于总线设备信息初始化晚于 i2c_adapter 初始化的情况，如 camer sensor 的 v4l2 子设备；而方法二则是用于较早初始化的情况。

4. 总线传输接口

总线传输是完成功能型事务的基础，之前已经了解到，一次总线传输可以进行多个 i2c_msg 的交互，i2c_msg 是传输的基本信息。下面来看一下详细内容：

```
    struct i2c_msg {
        //从地址
        __u16 addr;     /* slave address */
        //表示读/写,以及地址位数等属性
        __u16 flags;
        //信息长度
        __u16 len;        /* msg length */
        //信息数据
```

```
    __u8 *buf;          /* pointer to msg data */
};
```

标准 I²C 传输的接口函数是 i2c_transfer，具体内容如下：

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    unsigned long orig_jiffies;
    int ret, try;
    //使用 master_xfer 进行传输操作
    if(adap->algo->master_xfer) {
        //进行必要的锁操作
        if(in_atomic() || irqs_disabled()) {
            ret = i2c_trylock_adapter(adap);
            //原子操作和关中断不能等待长时间的操作,如果无法获得锁
            //则报告错误
            if(!ret)
                /* I2C activity is ongoing. */
                return -EAGAIN;
        } else {
            i2c_lock_adapter(adap);
        }

        /* Retry automatically on arbitration loss */
        orig_jiffies = jiffies;
        //重试机制
        for(ret = 0, try = 0; try <= adap->retries; try++) {
            //进行实际的传输
            ret = adap->algo->master_xfer(adap, msgs, num);
            //没有重试异常则跳出重试
            if(ret != -EAGAIN)
                break;
            //超时则跳出重试
            if(time_after(jiffies, orig_jiffies + adap->timeout))
                break;
        }
        //释放锁
        i2c_unlock_adapter(adap);

        return ret;
    } else {
        dev_dbg(&adap->dev, "I2C level transfers not supported\n");
```

```
                return -EOPNOTSUPP;
        }
    }
```

从代码中可见，实现一定的重传和超时机制主要还是通过 adapter 中操作接口 master_xfer 来完成的。

具体的操作通常由功能型驱动完成。下面还是以 mt9t111 sensor 为例：

```
    static int mt9t111_read_reg(struct i2c_client *client, u16 reg, u16 *val)
    {
        struct i2c_msg msg[1];
        u8 data[4];
        int err;

        //读寄存器时要先写入读取的寄存器地址,然后再进行读取操作
        //先写读取的寄存器地址信息
        msg->addr = client->addr;
        msg->flags = 0;
        msg->len = 2;
        msg->buf = data;
        data[0] = (reg & 0xff00) >> 8;
        data[1] = (reg & 0x00ff);
        err = i2c_transfer(client->adapter, msg, 1);
        if(err >= 0){
            //进行读操作
            msg->flags = I2C_M_RD;
            msg->len = 2; /* 2 byte read */
            err = i2c_transfer(client->adapter, msg, 1);
            if(err >= 0){
                *val = ((data[0] & 0x00ff) << 8)
                        | (data[1] & 0x00ff);
                return 0;
            }
        }
        return err;
    }

    static int mt9t111_write_reg(struct i2c_client *client, u16 reg, u16 val)
    {
        struct i2c_msg msg[1];
        u8 data[20];
        int err;
```

```
        msg->addr = client->addr;
        msg->flags = 0;
        msg->len = 4;
        msg->buf = data;
        data[0] = (u8)((reg & 0xff00) >> 8);
        data[1] = (u8)(reg & 0x00ff);
        data[2] = (u8)((val & 0xff00) >> 8);
        data[3] = (u8)(val & 0x00ff);
        err = i2c_transfer(client->adapter, msg, 1);
        if(err < 0)
            return err;

        return 0;
}
```

以上分别是读和写的例子，可见主要的工作就是通过 i2c_client 的信息构建 i2c_msg，然后调用 i2c_transfer 来完成传输。

为了免去驱动构建 i2c_msg，框架提供了以下接口以方便驱动调用：

```
int i2c_master_send(struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

这些函数都是对 i2c_ transfer 的封装。Linux 的 I^2C 框架主要提供的是主设备的功能，而从设备功能并没有涉及，这与使用 Linux 设备通常都通过 I^2C 总线连接从设备相关。

7.1.3 TI 芯片 I^2C 总线驱动相关实现详解

DM 3730 的 I^2C 控制器框架如图 7-4 所示。图 7-4 引自《DM 3730 芯片手册》中第 2798 页的框图。

从图 7-4 可见，作为 I^2C 控制器即可以做 master 也可以做 slave，在 Linux 内核中主要是实现 master 的功能；FIFO 用于进行数据缓冲，可以通过处理器读写，也可以通过 DMA 读写。

关于 DM 3730 I^2C 的驱动部分，主要分析相关初始化和总线传输的操作。

1. 初始化

先来看看初始化部分，作为 SoC 片内的控制器，相应的驱动都是作为 platform driver 存在的，在系统初始化时会注册相应的 platform device，而在 platform driver 的 probe 函数中则会根据 platform device 的信息进行初始化。这个初始化是十分重要的，现在就来详细了解其内容。

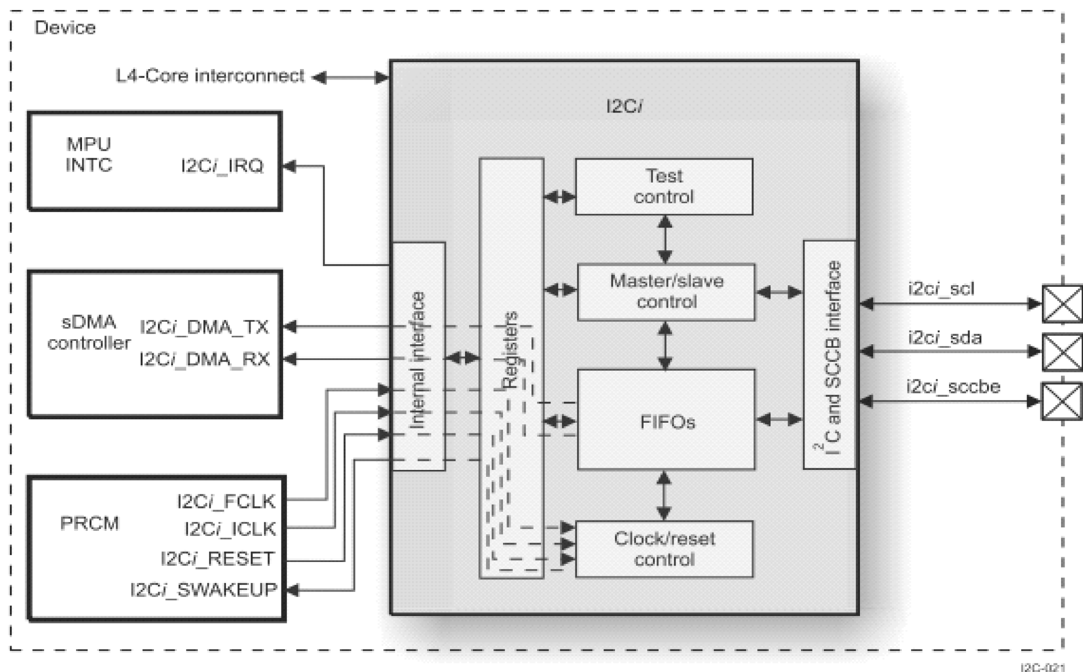


图 7-4　DM 3730 I^2C 控制器框图

```
static int __devinit omap_i2c_probe(struct platform_device  * pdev)
{
      struct omap_i2c_dev        * dev;
      struct i2c_adapter        * adap;
      struct resource              * mem, * irq, * ioarea;
      struct omap_i2c_bus_platform_data  * pdata = pdev -> dev. platform_data;
      irq_handler_t isr;
      int r;
      u32 speed = 0;

      /*  NOTE: driver uses the static register mapping  */
      //获得寄存器空间信息
      mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
...
      //获得中断信息
      irq = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
...
      //在内核的地址空间管理中标记寄存器空间使用.
      ioarea = request_mem_region(mem -> start, resource_size(mem),       pdev -> name);
...
      //分配管理实体
      dev = kzalloc(sizeof(struct omap_i2c_dev), GFP_KERNEL);
```

```
...
    //DM 3730 特殊操作实体，明确总线速度和处理器的唤醒延时
    if(pdata! = NULL){
        speed = pdata -> clkrate;
        dev -> set_mpu_wkup_lat = pdata -> set_mpu_wkup_lat;
    } else {
        speed = 100;      /* Default speed */
        dev -> set_mpu_wkup_lat = NULL;
    }

    //向管理实体中的属性赋值
    dev -> speed = speed;
    dev -> idle = 1;
    dev -> dev = &pdev -> dev;
    dev -> irq = irq -> start;
    //寄存器空间映射到内核虚拟空间
    dev -> base = ioremap(mem -> start, resource_size(mem));
    if(!dev -> base){
        r = -ENOMEM;
        goto err_free_mem;
    }
    //设置 platform device 的私有数据
    platform_set_drvdata(pdev, dev);

    //设置寄存器操作的相关信息，包括寄存器的地址间隔，以及不同寄存器顺序
    if(cpu_is_omap7xx())
        dev -> reg_shift = 1;
    else if(cpu_is_omap44xx() || cpu_is_ti81xx())
        dev -> reg_shift = 0;
    else
        dev -> reg_shift = 2;

    if(cpu_is_omap44xx() || cpu_is_ti81xx())
        dev -> regs = (u8 *) omap4_reg_map;
    else
        dev -> regs = (u8 *) reg_map;

    //enable 设备 runtime pm 功能
    pm_runtime_enable(&pdev -> dev);
    //激活 I2C 控制器使其处于 active 状态
    omap_i2c_unidle(dev);
    //读取硬件版本
```

```
    dev -> rev = omap_i2c_read_reg(dev, OMAP_I2C_REV_REG) & 0xff;
    //记录可能的勘误
    if(dev -> rev < = OMAP_I2C_REV_ON_3430)
        dev -> errata |= I2C_OMAP3_1P153;

    //根据版本获得 fifo size 并计算延时
    if(! (cpu_class_is_omap1() || cpu_is_omap2420())) {
        u16 s;

        /* Set up the fifo size - Get total size */
        s = (omap_i2c_read_reg(dev, OMAP_I2C_BUFSTAT_REG) >> 14) & 0x3;
        dev -> fifo_size = 0x8 << s;

        if(dev -> rev >= OMAP_I2C_REV_ON_4430) {
            dev -> fifo_size = 0;
            dev -> b_hw = 0; /* Disable hardware fixes */
        } else {
            dev -> fifo_size = (dev -> fifo_size / 2);
            dev -> b_hw = 1; /* Enable hardware fixes */
        }
        /* calculate wakeup latency constraint for MPU */
        if(dev -> set_mpu_wkup_lat! = NULL)
            dev -> latency = (1000000 * dev -> fifo_size) /
                          (1000 * speed / 8);
    }

    /* reset ASAP, clearing any IRQs */
    //reset I2C 控制器,清除中断,保证控制器处于已知状态
    omap_i2c_init(dev);

    //设置中断处理函数
    isr = (dev -> rev < OMAP_I2C_REV_2) ? omap_i2c_rev1_isr : omap_i2c_isr;
    r = request_irq(dev -> irq, isr, 0, pdev -> name, dev);
…
    //允许设备 idle
    omap_i2c_idle(dev);

    //向 I2C 框架注册 I2C 适配器实体
    adap = &dev -> adapter;
    i2c_set_adapdata(adap, dev);
    adap -> owner = THIS_MODULE;
    adap -> class = I2C_CLASS_HWMON;
```

```
        strlcpy(adap->name, "OMAP I2C adapter", sizeof(adap->name));
        //设备相关的 I2C 传输操作接口实体
        adap->algo = &omap_i2c_algo;
        adap->dev.parent = &pdev->dev;

        //总线控制器编号设置为 I2C 控制器编号
        adap->nr = pdev->id;
        //指定编号注册
        r = i2c_add_numbered_adapter(adap);
...
        return 0;

err_free_irq:
...
        return r;
}
```

probe 中需要的 platform device 是在 omap2_i2c_add_bus 中进行初始化的，主要内容如下：

```
static inline int omap2_i2c_add_bus(int bus_id)
{
        int l;
        struct omap_hwmod *oh;
        struct omap_device *od;
        char oh_name[MAX_OMAP_I2C_HWMOD_NAME_LEN];
        struct omap_i2c_bus_platform_data *pdata;

        //引脚设置
        omap2_i2c_mux_pins(bus_id);
        //获得 I2C 控制器名
        l = snprintf(oh_name, MAX_OMAP_I2C_HWMOD_NAME_LEN, "i2c%d", bus_id);
        //获得相应的硬件模块信息
        oh = omap_hwmod_lookup(oh_name);
        if(!oh){
                pr_err("Could not look up %s\n", oh_name);
                return -EEXIST;
        }

        pdata = &i2c_pdata[bus_id - 1];
...
        if(cpu_is_omap34xx())
```

```
        pdata -> set_mpu_wkup_lat = omap_pm_set_max_mpu_wakeup_lat_compat;
    //这里会创建 platform device 并注册
    od = omap_device_build( name, bus_id, oh, pdata,
        sizeof( struct omap_i2c_bus_platform_data) ,
        omap_i2c_latency, ARRAY_SIZE( omap_i2c_latency) , 0) ;
    WARN( IS_ERR( od) , "Could not build omap_device for %s\n" , name) ;

    return PTR_ERR( od) ;
}
```

这里是与 I²C 控制器相关的初始化，总线设备的初始化是由 DM 3730 提供相应的接口，通过 omap_register_i2c_bus 来实现的，具体细节如下：

```
int __init omap_register_i2c_bus( int bus_id, u32 clkrate,
                        struct i2c_board_info const  * info, unsigned len)
{
    int err;

    if( info) {
        //这里通过 I²C 框架注册静态的 i2c_board_info,后续注册 adapter 时
        //会根据这些信息创建总线设备管理实体
        err = i2c_register_board_info( bus_id, info, len) ;
        if( err)
            return err;
    }
    //如果没有初始化总线频率,则使用传入的参数
    if( ! i2c_pdata[ bus_id - 1 ]. clkrate)
        i2c_pdata[ bus_id - 1 ]. clkrate = clkrate;

    i2c_pdata[ bus_id - 1 ]. clkrate & = ~ OMAP_I2C_CMDLINE_SETUP;
    //根据 I²C 控制器信息进行 platform device 的初始化
    return omap_i2c_add_bus( bus_id) ;
}
```

关于 I²C 总线的频率，TI 为其提供了启动参数设置的接口，可以在 bootargs 中加入命令 i2c_bus = bus_id,clkrate(in kHz)来进行总线频率的设置，具体的总线频率设置是在 omap_i2c_init 中进行的。提供参数设置的好处是：一些硬件问题可以通过调整总线频率验证或者解决，有了参数设置接口，就可以避免代码的修改和编译等费时的工作。

2. 总线传输

总线传输是由 omap_i2c_algo 描述的，细节如下：

```
static const struct i2c_algorithm omap_i2c_algo = {
    . master_xfer  = omap_i2c_xfer,
```

```
    .functionality = omap_i2c_func,
};
```

主要是传输接口 omap_i2c_xfer，下面对传输过程进行详细分析：

```
static int omap_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[], int num)
{
    struct omap_i2c_dev *dev = i2c_get_adapdata(adap);
    int i;
    int r;
    //保证控制器 active
    omap_i2c_unidle(dev);
    //等待总线空闲
    r = omap_i2c_wait_for_bb(dev);
    if(r<0)
        goto out;
    //对所有的 msg 进行传输
    for(i=0; i<num; i++){
        //根据是否是最后一个 msg,表示是否加 stop condition
        r = omap_i2c_xfer_msg(adap, &msgs[i],(i==(num - 1)));
        if(r!=0)
            break;
    }

    if(r==0)
        r = num;
    //等待总线空闲则结束
    omap_i2c_wait_for_bb(dev);
out:
    omap_i2c_idle(dev);
    return r;
}
```

msg 的传输是通过 omap_i2c_xfer_msg 来完成的，下面来看看具体实现过程：

```
static int omap_i2c_xfer_msg(struct i2c_adapter *adap, struct i2c_msg *msg, int stop)
{
    //首先获得相应的 I2C 控制器的管理实体
    struct omap_i2c_dev *dev = i2c_get_adapdata(adap);
    int r;
    u16 w;

    if(msg->len==0)
```

```
        return -EINVAL;
//传输操作的从设备地址写入寄存器
omap_i2c_write_reg(dev, OMAP_I2C_SA_REG, msg->addr);

//将 msg 的数据信息和长度转给驱动做后续的处理
dev->buf = msg->buf;
dev->buf_len = msg->len;
//写 payload 长度计数寄存器
omap_i2c_write_reg(dev, OMAP_I2C_CNT_REG, dev->buf_len);

/* Clear the FIFO Buffers */
//操作之前,需要把 FIFO 清空
w = omap_i2c_read_reg(dev, OMAP_I2C_BUF_REG);
w |= OMAP_I2C_BUF_RXFIF_CLR | OMAP_I2C_BUF_TXFIF_CLR;
omap_i2c_write_reg(dev, OMAP_I2C_BUF_REG, w);

//初始化 completion,用于操作同步
init_completion(&dev->cmd_complete);
dev->cmd_err = 0;

//enable 控制器,设置 master 和 start condition(总线传输起始标记)
w = OMAP_I2C_CON_EN | OMAP_I2C_CON_MST | OMAP_I2C_CON_STT;

/* High speed configuration */
//总线速度大于 400K 就是 HS 模式
if(dev->speed > 400)
        w |= OMAP_I2C_CON_OPMODE_HS;
//如果 10bit 地址需要进行设置
if(msg->flags & I2C_M_TEN)
        w |= OMAP_I2C_CON_XA;
//如果是写操作,需要设置为 transmit mode
if(!(msg->flags & I2C_M_RD))
        w |= OMAP_I2C_CON_TRX;

//没有硬件 bug 的芯片,在最后一个 msg 需要设置 stop condition
//用于结束时控制器发送传输结束标记
if(!dev->b_hw && stop)
        w |= OMAP_I2C_CON_STP;

//将上述总线操作特性写入控制寄存器,设置了控制器 enable 就开始与
//从设备进行通信了
```

```
omap_i2c_write_reg(dev, OMAP_I2C_CON_REG, w);
//某些芯片是不能同时设置 STT 和 STP 的,这样在结尾 msg 时需要分开设置
if(dev->b_hw && stop){
    unsigned long delay = jiffies + OMAP_I2C_TIMEOUT;
    u16 con = omap_i2c_read_reg(dev, OMAP_I2C_CON_REG);
    //STT 会在 start condition 产生后被清除,这里等待被清除
    while(con & OMAP_I2C_CON_STT){
        con = omap_i2c_read_reg(dev, OMAP_I2C_CON_REG);

        /* Let the user know if i2c is in a bad state */
        if(time_after(jiffies, delay)){
            dev_err(dev->dev, "controller timed out "
            "waiting for start condition to finish\n");
            return -ETIMEDOUT;
        }
        cpu_relax();
    }
    //单独设置 STP 来产生 stop condition
    w |= OMAP_I2C_CON_STP;
    w &= ~OMAP_I2C_CON_STT;
    omap_i2c_write_reg(dev, OMAP_I2C_CON_REG, w);
}

if(dev->set_mpu_wkup_lat != NULL)
    dev->set_mpu_wkup_lat(dev->dev, dev->latency);
//等待操作完成
r = wait_for_completion_timeout(&dev->cmd_complete, OMAP_I2C_TIMEOUT);
if(dev->set_mpu_wkup_lat != NULL)
      dev->set_mpu_wkup_lat(dev->dev, -1);
//数据长度改为 0
dev->buf_len = 0;

if(r < 0)
    return r;
//超时则 reset 控制器
if(r == 0){
    dev_err(dev->dev, "controller timed out\n");
    omap_i2c_init(dev);
    return -ETIMEDOUT;
}
//正常则返回
if(likely(!dev->cmd_err))
```

```
            return 0;

        /* We have an error */
        //出现 overflow 或者 underrun 等错误则 reset 控制器
    if(dev->cmd_err & (OMAP_I2C_STAT_AL | OMAP_I2C_STAT_ROVR | OMAP_I2C_STAT_
    XUDF)){
            omap_i2c_init(dev);
            return -EIO;
        }
        //没有等到 ACK 则特别处理
        if(dev->cmd_err & OMAP_I2C_STAT_NACK){
            //如果忽略 nak 则直接返回
            if(msg->flags & I2C_M_IGNORE_NAK)
                return 0;
            //在有 stop condition 的情况重发 stop condition
            if(stop){
                w = omap_i2c_read_reg(dev, OMAP_I2C_CON_REG);
                w |= OMAP_I2C_CON_STP;
                omap_i2c_write_reg(dev, OMAP_I2C_CON_REG, w);
            }
            return -EREMOTEIO;
        }
        return -EIO;
    }
```

这里主要是根据 msg 的属性进行控制器的基本设置，实际的数据操作在中断处理函数中执行。详细分析如下：

```
    static irqreturn_t omap_i2c_isr(int this_irq, void *dev_id)
    {
        struct omap_i2c_dev *dev = dev_id;
        u16 bits;
        u16 stat, w;
        int err, count = 0;

        if(dev->idle)
                return IRQ_NONE;
        //读取 enable 的中断
        bits = omap_i2c_read_reg(dev, OMAP_I2C_IE_REG);
        //处理所有已经 enable 并且上报事件的中断,这里相当于轮询操作
        while((stat = (omap_i2c_read_reg(dev, OMAP_I2C_STAT_REG))) & bits){
            dev_dbg(dev->dev, "IRQ(ISR = 0x%04x)\n", stat);
```

```
        //中断处理占用太长时间,需要退出
        if(count ++ == 100){
            dev_warn(dev -> dev, "Too much work in one IRQ\n");
            break;
        }

        err = 0;
complete:
        //先清除状态位,保证控制器更新状态,某些位需要后续清除
        omap_i2c_write_reg(dev, OMAP_I2C_STAT_REG, stat &
                ~(OMAP_I2C_STAT_RRDY | OMAP_I2C_STAT_RDR |
                OMAP_I2C_STAT_XRDY | OMAP_I2C_STAT_XDR));
        //NACK 问题则重发
        if(stat & OMAP_I2C_STAT_NACK){
            err |= OMAP_I2C_STAT_NACK;
            omap_i2c_write_reg(dev, OMAP_I2C_CON_REG, OMAP_I2C_CON_STP);
        }
        //特殊的错误进行记录
        if(stat & OMAP_I2C_STAT_AL){
            dev_err(dev -> dev, "Arbitration lost\n");
            err |= OMAP_I2C_STAT_AL;
        }
        //ARDY 状态表示操作完成,其他不可恢复错误的情况时,则报告完成操作
        //之前已经对错误进行记录
    if(stat &(OMAP_I2C_STAT_ARDY | OMAP_I2C_STAT_NACK | OMAP_I2C_STAT_AL)){
            //通知操作完成前进行后续的清除操作
            omap_i2c_ack_stat(dev, stat &
                (OMAP_I2C_STAT_RRDY | OMAP_I2C_STAT_RDR |
                OMAP_I2C_STAT_XRDY | OMAP_I2C_STAT_XDR));
            omap_i2c_complete_cmd(dev, err);
            return IRQ_HANDLED;
        }
        //进行数据接收操作
        if(stat &(OMAP_I2C_STAT_RRDY | OMAP_I2C_STAT_RDR)){
            u8 num_bytes = 1;
            //勘误的 workaround
            if(dev -> errata & I2C_OMAP_ERRATA_I207)
                i2c_omap_errata_i207(dev, stat);
            //根据 fifo 的情况来确定后续操作的 byte 数目
            if(dev -> fifo_size){
                //RRDY 说明还需要接收的数目大于阈值,则以阈值为单位接收
```

```
            if(stat & OMAP_I2C_STAT_RRDY)
                num_bytes = dev -> fifo_size;
            else    /* read RXSTAT on RDR interrupt */
                //receive drain 表示还应读取 byte 的数目
                num_bytes = (omap_i2c_read_reg(dev,
                              OMAP_I2C_BUFSTAT_REG) >> 8) & 0x3F;
        }
        //一次操作 fifo 中的 byte 数
        while(num_bytes){
            num_bytes --;
            //读出数据
            w = omap_i2c_read_reg(dev, OMAP_I2C_DATA_REG);
            //检查是否到设定的操作数据长度
            if(dev -> buf_len){
                //将读取的数据写入设定的空间
                *dev -> buf ++ = w;
                dev -> buf_len --;
                …
            } else {
                //预定操作而此时仍有数据要操作则根据中断情况提示
                if(stat & OMAP_I2C_STAT_RRDY)
                    dev_err(dev -> dev,
                            "RRDY IRQ while no data"
                                " requested\n");
                if(stat & OMAP_I2C_STAT_RDR)
                    dev_err(dev -> dev,
                            "RDR IRQ while no data"
                                " requested\n");
                break;
            }
        }
        //清除状态
        omap_i2c_ack_stat(dev,
              stat &(OMAP_I2C_STAT_RRDY | OMAP_I2C_STAT_RDR));
        //重新查询状态寄存器,再开始操作
        continue;
    }
    //这里是进行写的操作
    if(stat &(OMAP_I2C_STAT_XRDY | OMAP_I2C_STAT_XDR)){
        u8 num_bytes = 1;
        //根据 fifo 的状态设定一次操作的数目
        if(dev -> fifo_size){
```

```
            //XRDY 说明还需要传送的数目大于阈值,则以阈值为单位接收
            if(stat & OMAP_I2C_STAT_XRDY)
                num_bytes = dev->fifo_size;
            else    /* read TXSTAT on XDR interrupt */
                // transmit drain 表示还应发送 byte 的数目
                num_bytes = omap_i2c_read_reg(dev,
                              OMAP_I2C_BUFSTAT_REG) & 0x3F;
        }
        //进行一次操作
        while(num_bytes){
            num_bytes --;
            w = 0;
            //检查是否到设定的操作数据长度
            if(dev->buf_len){
                //准备将设定空间的数据写入寄存器
                w = *dev->buf++;
                dev->buf_len --;
                …
            } else {
                //预定操作而此时仍有数据要操作则根据中断情况提示
                if(stat & OMAP_I2C_STAT_XRDY)
                    dev_err(dev->dev,
                          "XRDY IRQ while no "
                          "data to send\n");
                if(stat & OMAP_I2C_STAT_XDR)
                    dev_err(dev->dev,
                          "XDR IRQ while no "
                          "data to send\n");
                break;
            }
            …
            //传送的数据写入寄存器
            omap_i2c_write_reg(dev, OMAP_I2C_DATA_REG, w);
        }
        //清除状态
        omap_i2c_ack_stat(dev,
              stat &(OMAP_I2C_STAT_XRDY | OMAP_I2C_STAT_XDR));
        //重新查询状态寄存器,再开始操作
        continue;
    }
    //其他异常进行标记
    if(stat & OMAP_I2C_STAT_ROVR){
```

```
                dev_err(dev->dev, "Receive overrun\n");
                dev->cmd_err |= OMAP_I2C_STAT_ROVR;
            }
            if(stat & OMAP_I2C_STAT_XUDF){
                dev_err(dev->dev, "Transmit underflow\n");
                dev->cmd_err |= OMAP_I2C_STAT_XUDF;
            }
        }
        return count ? IRQ_HANDLED : IRQ_NONE;
    }
```

从代码中可见，主要的数据操作都是在中断中进行处理的。由于 I^2C 作为控制和获取状态信息，数据量并不大，而且相关的操作频率并不高，这样在中断中进行相关的处理，对系统的影响并不大。

在中断处理中涉及的 FIFO 阈值以及使能的中断都是在 omap_i2c_init 中进行操作的，相关代码如下：

```
if(dev->fifo_size){
    /* Note: setup required fifo size - 1. RTRSH and XTRSH */
    buf = (dev->fifo_size - 1) << 8 | OMAP_I2C_BUF_RXFIF_CLR |
        (dev->fifo_size - 1) | OMAP_I2C_BUF_TXFIF_CLR;
    omap_i2c_write_reg(dev, OMAP_I2C_BUF_REG, buf);
}
...
/* Enable interrupts */
dev->iestate = (OMAP_I2C_IE_XRDY | OMAP_I2C_IE_RRDY |    OMAP_I2C_IE_ARDY |
OMAP_I2C_IE_NACK | OMAP_I2C_IE_AL) | ((dev->fifo_size) ?    (OMAP_I2C_IE_RDR |
OMAP_I2C_IE_XDR) : 0);
omap_i2c_write_reg(dev, OMAP_I2C_IE_REG, dev->iestate);
```

这样整个的传输从设置到实际操作就完整了。

7.1.4 I^2C 总线驱动电源管理相关说明

关于 I^2C 总线的电源管理，首先来看看 bus_type 中描述的总线级别的电源管理操作 i2c_device_pm_ops，细节如下：

```
static const struct dev_pm_ops i2c_device_pm_ops = {
    .suspend = i2c_device_pm_suspend,
    .resume = i2c_device_pm_resume,
    .freeze = i2c_device_pm_freeze,
    .thaw = i2c_device_pm_thaw,
```

```
        .poweroff = i2c_device_pm_poweroff,
        .restore = i2c_device_pm_restore,
        SET_RUNTIME_PM_OPS(
            pm_generic_runtime_suspend,
            pm_generic_runtime_resume,
            pm_generic_runtime_idle
        )
    };
```

其中包括了 SLM 和 runtime pm 的操作接口，runtime pm 是标准接口，用于调用设备模型 device_driver 中相应的操作接口，通常并不定义。下面以 suspend 操作为例看看具体的操作细节：

```
    static int i2c_device_pm_suspend(struct device *dev)
    {
        //获得设备模型中驱动的 pm 操作接口
        const struct dev_pm_ops *pm = dev->driver ? dev->driver->pm : NULL;

        //如果设备模型中已经定义了操作,则以设备模型为主
        if(pm) {
            //检查设备状态,如果在 suspend 状态则返回,否则执行接口操作并返回
            if(pm_runtime_suspended(dev))
                return 0;
            else
                return pm->suspend ? pm->suspend(dev) : 0;
        }
        //执行 I2C 的 suspend 操作
        return i2c_legacy_suspend(dev, PMSG_SUSPEND);
    }
```

通常都会执行 I^2C 相关的操作，下面来看看具体的实现过程：

```
    static int i2c_legacy_suspend(struct device *dev, pm_message_t mesg)
    {
        //转换为总线设备
        struct i2c_client *client = i2c_verify_client(dev);
        struct i2c_driver *driver;

        if(!client || !dev->driver)
            return 0;
        //转换为总线驱动
        driver = to_i2c_driver(dev->driver);
        if(!driver->suspend)
```

```
        return 0;
    //执行驱动的 suspend 操作
    return driver->suspend(client, mesg);
}
```

可见最终需要执行 i2c_driver 的电源管理操作。

总线级别的电源管理通过 i2c_driver 中的电源管理接口进行实际操作，而对于 SLM 操作中的设备遍历则是在设备模型级别进行的。

控制器级别的电源管理通过 platform driver 来进行定义，具体到 DM 3730 就是 omap_i2c_driver，具体的实现中设备模型的实体 driver 中并没有定义 pm 相关的操作。这并不代表没有在控制器级别进行电源管理操作，控制器有 platform device 实体，在设备模型级别会执行 platform bus 的电源管理操作，其中就会进行实际的电源管理操作。DM 3730 相关的部分在 SoC 电源管理中会进行详细的分析说明。

这样 I^2C 总线的电源管理各个层面就完整了。

7.2 串行外设接口总线（SPI）

7.2.1 SPI 总线驱动需求

串行外围设备接口（Serial Peripheral Interface，SPI）是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。SPI 接口主要用于处理器与 EEPROM、FLASH、WLAN 等设备的连接。SPI 是一种高速、全双工、同步总线。连接方面共需要四根线，有线数少、为 PCB 的布局上节省空间的优点。SPI 速率通常介于 1 ~ 70 MHz 之间，字长范围可以设置从 4 ~ 32 bits。

SPI 以主从方式工作，通常有一个主设备和一个或多个从设备，有 4 个信号，分别是 MOSI、MISO、SCLK 和 CS 信号，下面分别介绍：

- MOSI(Master Out Slave In)：主器件数据输出，从器件数据输入。
- MISO(Master In Slave Out)：主器件数据输入，从器件数据输出。
- SCLK ：时钟信号，由主器件产生。
- CS：从器件使能信号，由主器件控制。

其中 CS 控制从设备是否被选中，只有片选信号为预先规定的使能信号时（高电位或低电位)，对应从设备的操作才有效。这样就允许单个 SPI 总线上连接多个 SPI 从设备。需要注意的是，单总线连接多个从设备时需要多个 CS 信号源，具体可以使用 GPIO 来代替控制器的 CS 信号（如果控制器信号不足)。对于 SPI 总线从设备并没有固化的地址，而是通过 CS 信号区分不同的设备。

SPI 总线数据传输信号规范如图 7-5 所示。图 7-5 引自《DM 3730 芯片手册》中第 2992 页框图。

从图 7-5 可见，信号的极性、相位等性质是可以配置的，而对于传输的主从双方也是可以同时进行数据传输的，这些功能都是必需的。另外因为 SPI 总线频率最高可以达到 70

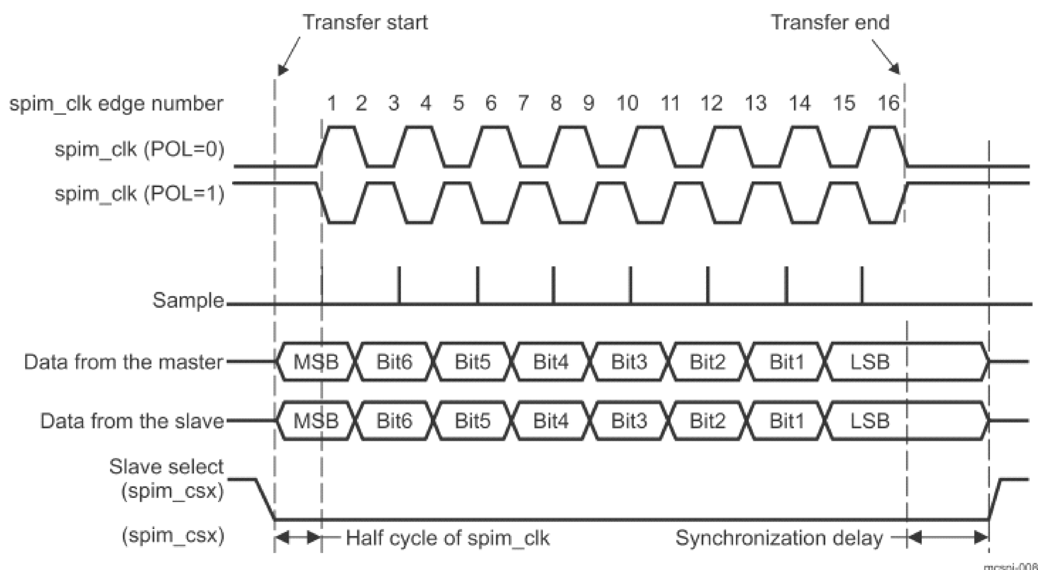


图 7-5　SPI 总线数据传输信号规范

MHz，所以需要较大吞吐量的操作方案进行数据传输，通常使用 DMA 进行操作。

总体上来说，系统对于 SPI 总线驱动的需求就是要能实现总线的各种功能，并且满足类似于 I^2C 总线的各种无关性需求。

7.2.2　SPI 总线驱动框架解析

1. 总线相关以及核心框架

同 I^2C 总线相同，先来了解 SPI 总线的实现。在介绍系统初始化的时候，整个 SPI 的初始化函数 spi_init 定义为 postcore_initcall，这样可保证 SPI 总线在 SPI 设备或者驱动注册之前进行注册，从而确保总线的功能。初始化中包含如下语句：

```
status = bus_register(&spi_bus_type);
```

这里注册了总线类型，其中 bus_register 设置总线自动 probe 功能。而 spi_bus_type 作为了解整个总线框架的入口，详细定义如下：

```
struct bus_type spi_bus_type = {
        .name           = "spi",
        .dev_attrs      = spi_dev_attrs,
        .match          = spi_match_device,
        .uevent         = spi_uevent,
        .suspend = spi_suspend,
        .resume         = spi_resume,
};
```

这里重点的接口是 match。下面详细分析该接口：

```
static int spi_match_device(struct device *dev, struct device_driver *drv)
{
        const struct spi_device     *spi = to_spi_device(dev);
        const struct spi_driver      *sdrv = to_spi_driver(drv);

        /* Attempt an OF style match */
        if(of_driver_match_device(dev, drv))
                return 1;

        //如果有支持的 id_table 则进行比较
        if(sdrv->id_table)
                return!!spi_match_id(sdrv->id_table, spi);

        //名字相同则匹配
        return strcmp(spi->modalias, drv->name) ==0;
}
```

从分析中可见，主要就是进行名字匹配。这里并没有像 I^2C 验证设备类型，这是为什么呢？难道总线控制器不是设备吗？总线控制器还是设备，但是 SPI 的实现方式不同，在初始化函数 spi_init 中有如下代码：

```
status = class_register(&spi_master_class);
```

这里创建了 class 功能类，总线控制器作为 class 设备进行管理，而 match 只是匹配相同 bus 的设备，这样总线控制器设备就不会进行 match 操作了。总线进行匹配的设备是 spi_device，这就是总线连接设备的管理实体。

在 spi_bus_type 中并没有看到 probe 接口，probe 操作是 match 之后就需要进行的操作，主要有设备特殊资源的探测、申请以及初始化的操作。SPI 总线是如何实现该功能的呢？下面来看看对应的 spi_driver 注册的操作：

```
int spi_register_driver(struct spi_driver *sdrv)
{
    sdrv->driver.bus = &spi_bus_type;
    if(sdrv->probe)
        sdrv->driver.probe = spi_drv_probe;
    if(sdrv->remove)
        sdrv->driver.remove = spi_drv_remove;
    if(sdrv->shutdown)
        sdrv->driver.shutdown = spi_drv_shutdown;
    return driver_register(&sdrv->driver);
}
```

这里直接将接口赋值给驱动的接口，在设备模型的统一操作中如果找不到总线相关的接口就会执行 driver 的接口，这样问题就解决了。与 I^2C 比较又是不同的实现方式。系统提供了统一的框架，具体的实现方法还是仁者见仁的。

SPI 总线框架对这两类设备进行管理，具体的框架如图 7-6 所示。

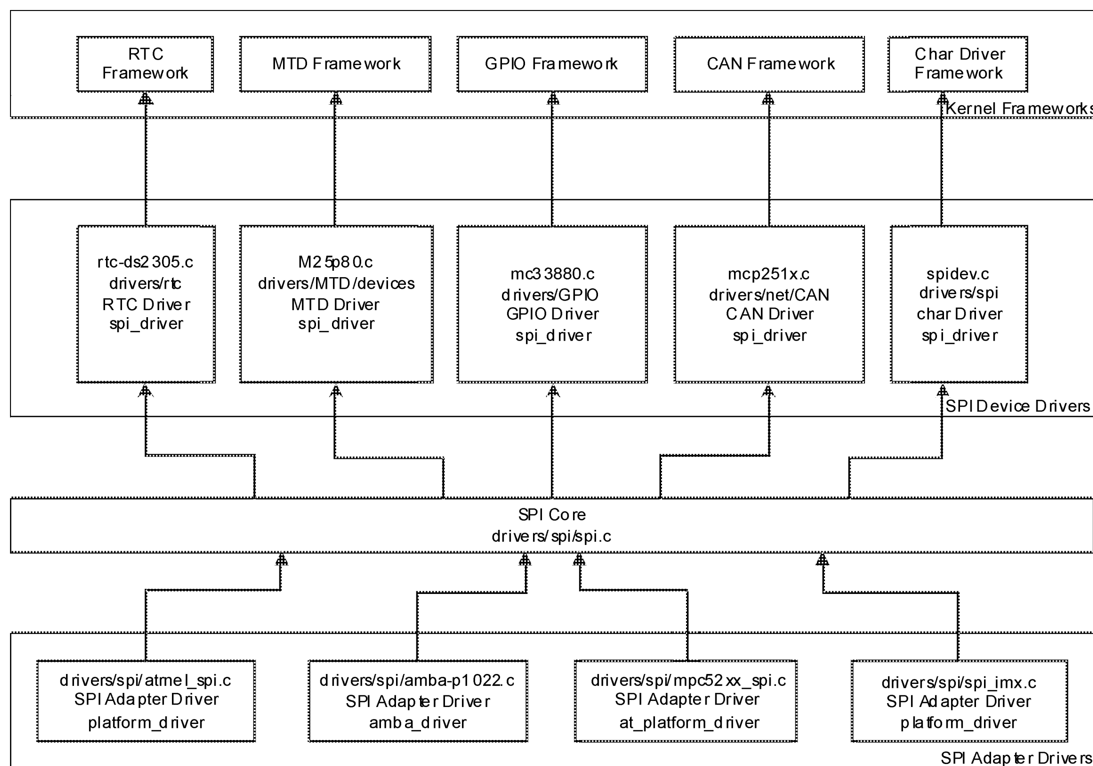


图 7-6　Linux SPI 总线框架

从图 7-6 中可见，SPI 设备主要是为内核中不同的功能模块服务的，当不同功能模块的驱动需要使用 SPI 设备进行操作时，通常会在具体的管理实体中嵌入 spi_device，实际的总线操作通过 spi_device 来实现。图中右侧可见特殊的 spidev，这是对应用开放的接口设备，SPI 通过 CS 信号进行设备管理，而相应的 CS 排列主要属于板级信息，所以不适合像 I^2C 总线一样将总线控制器对应用开放（应用程序操作设备需要对不同板子进行修改）。将 CS 信息对应用屏蔽，只开放具体的设备，这就是 spidev 的作用，应用只是针对具体的设备进行操作，即设备的应用层驱动（驱动在用户态执行），要在应用层使用该功能需要在注册设备时将 spi_board_info 中的 modalias 设置为 spidev，这样就可以在应用层实现相应设备的驱动。通常的 SPI 驱动都是在内核实现并在内核态执行的，所以就不对 spidev 进行详细的分析。

SPI 总线框架整体的功能就是这样，接下来看看具体各部分的实现。

2. 总线控制器相关

SPI 总线框架对总线控制器的管理主要是通过 i2c_adapter 来实现的，其详细分析如下：

```
struct spi_master {
    //设备模型相关
    struct device       dev;
    //框架管理的链表
    struct list_head list;

    //总线编号,通常与硬件控制器编号对应
    s16                 bus_num;
    //控制器拥有的片选数目
    u16                 num_chipselect;
    //控制器要求的 DMA 范围
    u16                 dma_alignment;

    //控制器特殊的属性,通常与信号极性、相位、CS 的极性相关
    u16                 mode_bits;

    //驱动的限制如半双工、只读或只写等
    u16                 flags;
    //相关的锁
    spinlock_t          bus_lock_spinlock;
    struct mutex        bus_lock_mutex;
    bool                bus_lock_flag;

    //该接口主要用于激活总线控制器与特定设备的总线连接,需要对传输参数做匹配
    int                 ( * setup)(struct spi_device  * spi);
    //该接口用于总线事务的传输
    int                 ( * transfer)(struct spi_device  * spi,      struct spi_message  * mesg);
    //当某个连接在控制器上的设备移除时需要进行相关操作的接口.
    void                ( * cleanup)(struct spi_device  * spi);
};
```

关于 setup 接口，框架为设备进行操作提供了统一的接口 spi_setup，其中会对信号属性、传输字长、总线频率等进行匹配和设置。

这里所有的接口都是与设备总线控制器相关的，通过这些接口设备的总线控制器实现总线操作。

SPI 总线框架为总线控制器驱动开发提供了相关的接口，具体如下：

```
//分配总线控制器管理实体
struct spi_master  * spi_alloc_master(struct device  * dev, unsigned size);
//注册加入系统管理
int spi_register_master(struct spi_master  * master);
//注销相关总线控制器管理实体
```

```
void spi_unregister_master(struct spi_master *master);
//通过总线编号获得总线控制器管理实体
struct spi_master *spi_busnum_to_master(u16 busnum);
```

下面详细看看 spi_alloc_master 的具体实现过程：

```
struct spi_master *spi_alloc_master(struct device *dev, unsigned size)
{
    struct spi_master        *master;

    if(!dev)
        return NULL;
    //分配 spi_master 以及设备指定所需要的空间
    master = kzalloc(size + sizeof *master, GFP_KERNEL);
    if(!master)
        return NULL;
    //设备模型初始化
    device_initialize(&master->dev);
    //指定为 spi_master_class
    master->dev.class = &spi_master_class;
    master->dev.parent = get_device(dev);
    //将为设备分配的空间(在 spi_master 之后)写入 dev data 中
    spi_master_set_devdata(master, &master[1]);

    return master;
}
```

这里还为设备分配了所需的空间，设备不需要进行相应的操作，只要通过 spi_master_get_devdata 来获得相应的指针即可。

spi_register_master 注册了总线控制器，就相当于建立了总线，相应的会将总线中的设备创建并注册。这里主要通过如下操作完成相应功能：

```
list_for_each_entry(bi, &board_list, list)
        spi_match_master_to_boardinfo(master, &bi->board_info);
```

这里是实现总线设备发现相关功能，根据板级的信息注册 spi_device。

总线的主要功能（包括总线传输事务、总线设备发现）就都在代表 SPI 总线的 SPI 总线控制器部分有了完整的支持。

3. 总线设备相关

对 SPI 总线设备的管理，SPI 总线框架提供了 spi_device 来完成该功能，详细内容如下：

```
struct spi_device {
    //设备模型相关
    struct device           dev;
    //所关联的总线控制器
    struct spi_master       * master;
    //设备要求的最大总线频率
    u32                     max_speed_hz;
    //所用的片选信号
    u8                      chip_select;
    //信号的模式,主要是相位、极性等
    u8                      mode;
    //字长信息
    u8                      bits_per_word;
    //设备为系统提供的中断号,通常是 GPIO 对应的中断号
    int                     irq;
    //总线控制器相关的状态信息和数据,由于每个设备的信号、字长等属性不同
    //所以需要保存这些状态,以便驱动进行总线传输时在不同的状态间切换.
    void                    * controller_state;
    void                    * controller_data;
    //用于与 SPI driver 绑定的名字.
    char                    modalias[SPI_NAME_SIZE];
};
```

从相关属性来看，主要是 master 及其与总线信号相关的属性。下面来看看与之关联 driver 的详细信息：

```
struct spi_driver {
    //表示支持的设备
    const struct spi_device_id  * id_table;
    //标准的设备模型操作,用于设备绑定的初始化以及设备移除的资源释放
    int             ( * probe)(struct spi_device  * spi);
    int             ( * remove)(struct spi_device  * spi);
    //电源管理相关接口
    void            ( * shutdown)(struct spi_device  * spi);
    int             ( * suspend)(struct spi_device  * spi, pm_message_t mesg);
    int             ( * resume)(struct spi_device  * spi);
    //设备模型接口
    struct device_driver        driver;
};
```

从 spi_driver 的内容分析可见，其主要负责总线相关以及电源管理相关的操作，操作接口中并不涉及实际的信息传输，可以说该驱动同样是处理总线和设备模型相关的事务，并不涉及功能的事务。功能型事务涉及操作的具体内容是在功能型驱动中通过 SPI 的接口实现

的。而功能型的事务需要有总线设备的信息。这就要求功能型驱动管理实体中能够包含 spi_device 的信息。这个桥梁是与 SPI 总线事务相关的，所以该工作自然就由 spi_driver 来实现，具体的接口就是 probe，这样各种实体就关联起来。以 ads7846_probe 接口为例，ads7846 是 SPI 总线接口的触屏控制器，在功能类型中对应于输入设备，相应的 spi_driver_的 probe 接口即 ads7846_probe 中有如下代码：

```
static int __devinit ads7846_probe(struct spi_device *spi)
{
    struct ads7846 *ts;
    struct ads7846_packet *packet;
    struct input_dev *input_dev;
    struct ads7846_platform_data *pdata = spi->dev.platform_data;
    unsigned long irq_flags;
    int err;
...
    /* don't exceed max specified sample rate */
    if(spi->max_speed_hz > (125000 * SAMPLE_BITS)) {
        dev_dbg(&spi->dev, "f(sample) %d KHz? \n",
                (spi->max_speed_hz/SAMPLE_BITS)/1000);
        return -EINVAL;
    }
    spi->bits_per_word = 8;
    spi->mode = SPI_MODE_0;
    err = spi_setup(spi);
    if(err < 0)
        return err;

    ts = kzalloc(sizeof(struct ads7846), GFP_KERNEL);
    packet = kzalloc(sizeof(struct ads7846_packet), GFP_KERNEL);
    input_dev = input_allocate_device();
    if(!ts || !packet || !input_dev) {
        err = -ENOMEM;
        goto err_free_mem;
    }

    dev_set_drvdata(&spi->dev, ts);

    ts->packet = packet;
    ts->pdata = pdata;
    ts->spi = spi;
    ts->input = input_dev;
    ts->vref_mv = pdata->vref_mv;
```

```
        ts -> swap_xy = pdata -> swap_xy;
    …
    }
```

从中可见，spi device、触屏设备管理实体以及 input device 建立了关联，这样从功能型设备到总线设备就形成了一个整体，可以进行完整的操作。只有总线设备与功能管理实体关联后才能完成功能性的工作。

无论 spi_device 还是 spi_driver 都属于 SPI 总线框架管理的运行时实体，而 SPI 总线设备通常是直接焊接在板子上面，这些固定的信息同样需要进行表示，在创建动态管理的设备实体时需要这些信息。在 SPI 总线框架中相应的实体是 spi_board_info，内容如下：

```
struct spi_board_info {
    //匹配的名字
    char          modalias[SPI_NAME_SIZE];
    //设备特殊数据
    const void     * platform_data;
    void          * controller_data;
    //设备连接的中断号,通常 GPIO 对应的中断
    int           irq;
    //设备能接受的总线频率
    u32           max_speed_hz;
    //设备所在总线编号
    u16           bus_num;
    //设备所在总线的片选信号
    u16           chip_select;
    //信号的模式
    u8            mode;
};
```

其中的主要信息是与总线信号相关的属性。

对 SPI 总线设备的创建，框架提供了相应的函数接口，具体如下：

```
struct spi_device * spi_new_device(struct spi_master * master, struct spi_board_info * chip)
```

从中可见，要创建总线设备需要 spi_master 和 spi_board_info，对于特定总线设备有 spi_board_info 信息，这就需要明确与 spi_master 关联。

创建总线设备有两种方法：

方法一是通过 struct spi_master * spi_busnum_to_master(u16 busnum)；来获得指定的 spi_master，然后调用 spi_new_device 来创建总线设备。

方法二是早期的初始化阶段调用 spi_register_board_info(struct spi_board_info const * info, unsigned n);函数注册某个 SPI 总线上的所有设备信息，在总线控制器 spi_master 会在注册时遍历这些信息为总线上的设备创建 spi_device。

方法一主要用于总线设备信息初始化晚于 spi_master 初始化的情况，如使用 SPI 总线的

V4L2 子设备；而方法二则用于较早初始化的情况。

4. 总线传输接口

总线传输是完成功能型事务的基础。SPI 总线传输是由 spi_message 和 spi_transfer 共同进行管理的，下面来看一下详细内容：

```
struct spi_message {
    //spi_transfer 的链表
    struct list_head        transfers;
    //关联的 SPI 总线设备
    struct spi_device       * spi;
    //是否已经进行了 DMA 映射
    unsigned            is_dma_mapped:1;

    /* completion is reported through a callback */
    //完成完整传输后报告的回调
    void                ( * complete)( void  * context);
    //回调的参数,通常是 completion
    void                * context;
    //实际成功传输的长度以 byte 为单位
    unsigned            actual_length;
    //传输状态,0 为成功,负数为错误值
    int                 status;
    //总线控制器驱动使用,主要用于总线传输队列
    struct list_head        queue;
    void                * state;
};

struct spi_transfer {
    //数据空间
    const void      * tx_buf;
    void        * rx_buf;
    //数据长度
    unsigned   len;
    //如果使用 DMA,相关的数据地址信息
    dma_addr_t      tx_dma;
    dma_addr_t      rx_dma;
    //transfer 完成是否需要 CS 变化
    unsigned   cs_change:1;
    //字长
    u8          bits_per_word;
    //从数据操作完到 CS 变化的延时设置
```

```
        u16             delay_usecs;
        //总线频率
        u32             speed_hz;
        //传输列表
        struct list_head transfer_list;
    };
```

可见 SPI 总线设备的总线传输是以 spi_transfer 为单位的，可以进行批处理，将一次批处理所有的 spi_transfer，并形成链表通过 spi_message 进行管理。而 spi_message 中会与具体的 spi_device 关联，这样某个 SPI 总线设备的总线传输需要的所有信息就可以通过 spi_message 进行管理。

通常设备进行 SPI 传输的操作流程如下：

```
//初始化 spi_message
spi_message_init(m);
//设置传输 transfer 参数
packet->read_y_cmd[0] = READ_Y(vref);
packet->read_y_cmd[1] = 0;
packet->read_y_cmd[2] = 0;
x->tx_buf = &packet->read_y_cmd[0];
x->rx_buf = &packet->tc.y_buf[0];
x->len = 3;
//加入链表
spi_message_add_tail(x, m);
```

功能型驱动可以根据自身的需要设置 spi_transfer，读写操作信息都设置在一起，然后通过 spi_message_add_tail 加入 spi_message 链表中；在 spi_message 准备好之后，就可以通过 spi_sync 进行实际的传输操作。具体如下：

```
error = spi_sync(ts->spi, m);
```

这样设备就同步等待操作完成，具体的实现是通过__spi_sync 来完成的，细节如下：

```
static int __spi_sync(struct spi_device *spi, struct spi_message *message, int bus_locked)
{
    //声明 completion
    DECLARE_COMPLETION_ONSTACK(done);
    int status;
    //获得总线控制器
    struct spi_master *master = spi->master;
    //同步操作回调和参数
    message->complete = spi_complete;
```

```
        message -> context = &done;

        //总线独享控制
        if( ! bus_locked)
            mutex_lock( &master -> bus_lock_mutex) ;
        //异步锁操作,实际调用的总线控制器的 transfer 操作直接返回
        status = spi_async_locked( spi, message) ;

        if( ! bus_locked)
            mutex_unlock( &master -> bus_lock_mutex) ;
        //等待总线控制器操作完成
        if( status == 0) {
            wait_for_completion( &done) ;
            status = message -> status;
        }
        message -> context = NULL;
        return status;
    }
```

可见主要操作是由总线控制器驱动完成的，而在总线框架层则是等待 completion 同步操作完成后返回，这样就可保证整个传输的操作是以同步方式实现的。

7.2.3 TI 芯片 SPI 总线驱动相关实现详解

DM 3730 的 SPI 控制器框架如图 7-7 所示。图 7-7 引自《DM 3730 芯片手册》中第 2998 页的框图。

从图 7-7 可见，作为 SPI 控制器既可以做 master 也可以做 slave，在 Linux 内核中主要是实现 master 的功能；FIFO 用于进行数据缓冲，可以通过处理器读写，也可以通过 DMA 读写，由于 SPI 总线速度要高，所以当进行大量数据读写时需要使用 DMA 进行操作。另外不同的 channel 代表不同 CS 控制的设备，这样可以对设备进行单独的操作。

对于 DM 3730 SPI 的驱动部分，主要分析相关的初始化和总线传输的操作。

1. 初始化

先来看看初始化部分，SPI 总线控制器作为 SoC 片内的控制器，相应的驱动主要作为 platform driver 存在，在系统初始化时会注册相应的 platform device（通过 omap_init_mcspi 来实现），而在 platform driver 的 probe 函数中会根据 platform device 的信息进行初始化，这个初始化十分重要。下面详细了解其内容：

```
    static int __init omap2_mcspi_probe( struct platform_device  * pdev)
    {
        struct spi_master        * master;
        struct omap2_mcspi            * mcspi;
        struct resource               * r;
```

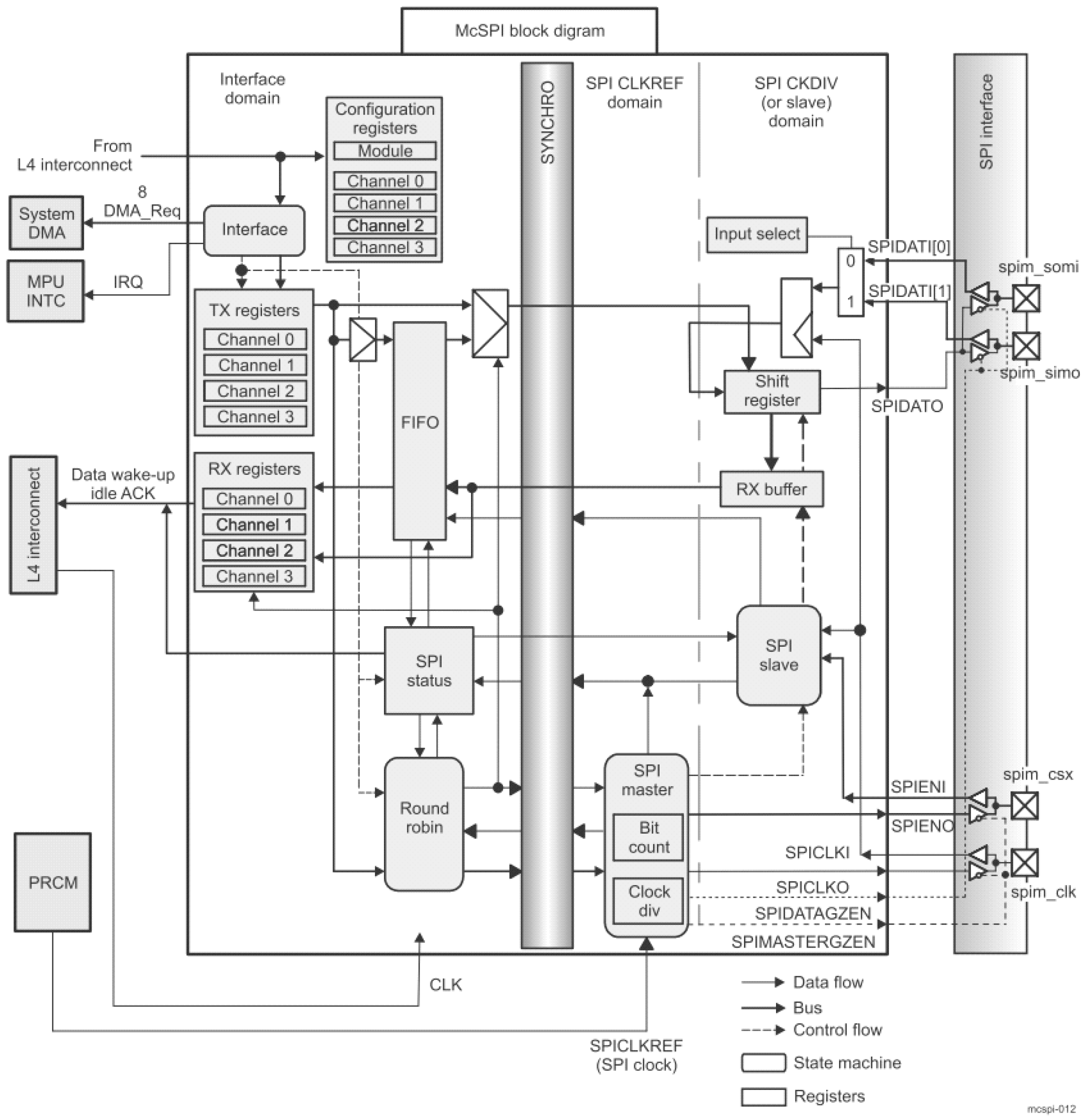


图 7-7　DM 3730 SPI 总线控制器框架

```
int                         status = 0, i;
const u8              *rxdma_id, *txdma_id;
unsigned              num_chipselect;
//根据 SPI 控制器的编号设置 DMA request 信号的 ID,以及拥有的 CS 信号数目
switch(pdev->id) {
case 1:
      rxdma_id = spi1_rxdma_id;
      txdma_id = spi1_txdma_id;
      num_chipselect = 4;
      break;
```

```
        case 2:
            rxdma_id = spi2_rxdma_id;
            txdma_id = spi2_txdma_id;
            num_chipselect = 2;
            break;
...
        default:
            return - EINVAL;
        }
        //分配总线控制器管理实体,并为设备独立管理实体分配空间
        master = spi_alloc_master(&pdev -> dev, sizeof *mcspi);
        if(master == NULL) {
            dev_dbg(&pdev -> dev, "master allocation failed\n");
            return - ENOMEM;
        }

        /* the spi -> mode bits understood by this driver: */
        //设置总线控制器支持的信号属性
        master -> mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
        //设置总线编号
        if(pdev -> id! = -1)
            master -> bus_num = pdev -> id;
        //SPI 总线控制器操作接口初始化
        master -> setup = omap2_mcspi_setup;
        master -> transfer = omap2_mcspi_transfer;
        master -> cleanup = omap2_mcspi_cleanup;
        master -> num_chipselect = num_chipselect;
        //设置设备特殊信息
        dev_set_drvdata(&pdev -> dev, master);
        //与设备独立的总线控制器管理实体进行关联
        mcspi = spi_master_get_devdata(master);
        mcspi -> master = master;

        //检查寄存器空间,并向系统申请相关空间
        r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if(r == NULL) {
            status = -ENODEV;
            goto err1;
        }
        if(! request_mem_region(r -> start,(r -> end - r -> start) + 1,
                        dev_name(&pdev -> dev))) {
            status = -EBUSY;
```

```
        goto err1;
    }
    //进行寄存器空间映射操作
    mcspi->phys = r->start;
    mcspi->base = ioremap(r->start, r->end - r->start + 1);
    if(!mcspi->base){
        dev_dbg(&pdev->dev, "can't ioremap MCSPI\n");
        status = -ENOMEM;
        goto err1aa;
    }
    //初始化 work 用于对 SPI 传输进行操作
    INIT_WORK(&mcspi->work, omap2_mcspi_work);

    spin_lock_init(&mcspi->lock);
    //设备驱动对于传输操作维护的队列
    INIT_LIST_HEAD(&mcspi->msg_queue);
    INIT_LIST_HEAD(&omap2_mcspi_ctx[master->bus_num - 1].cs);

    //时钟相关的操作
    mcspi->ick = clk_get(&pdev->dev, "ick");
    if(IS_ERR(mcspi->ick)){
        dev_dbg(&pdev->dev, "can't get mcspi_ick\n");
        status = PTR_ERR(mcspi->ick);
        goto err1a;
    }
    mcspi->fck = clk_get(&pdev->dev, "fck");
    if(IS_ERR(mcspi->fck)){
        dev_dbg(&pdev->dev, "can't get mcspi_fck\n");
        status = PTR_ERR(mcspi->fck);
        goto err2;
    }
    //为控制器的 DMA channel 管理申请管理实体的空间,并初始化
    mcspi->dma_channels = kcalloc(master->num_chipselect,
            sizeof(struct omap2_mcspi_dma), GFP_KERNEL);

    if(mcspi->dma_channels == NULL)
        goto err3;

    for(i = 0; i < num_chipselect; i++){
        mcspi->dma_channels[i].dma_rx_channel = -1;
        mcspi->dma_channels[i].dma_rx_sync_dev = rxdma_id[i];
        mcspi->dma_channels[i].dma_tx_channel = -1;
```

```
            mcspi->dma_channels[i].dma_tx_sync_dev = txdma_id[i];
        }
        //控制器 reset 保证状态确定
        if(omap2_mcspi_reset(mcspi) < 0)
            goto err4;
        //向 SPI 总线框架注册控制器实体
        status = spi_register_master(master);
        if(status < 0)
            goto err4;

        return status;
    ...
    }
```

因为 SPI 对吞吐量要求比较高，所以每个控制器采用单独的 work 处理，而系统通常有多个 SPI 总线控制器，继续提高吞吐量的方法就是建立 work queue，这些是在 omap2_mcspi_init 初始化中进行的，细节如下：

```
    static int __init omap2_mcspi_init(void)
    {
        //创建单内核线程执行的 work queue
        omap2_mcspi_wq = create_singlethread_workqueue(
                                omap2_mcspi_driver.driver.name);
        if(omap2_mcspi_wq == NULL)
            return -1;
        //注册 platform driver
        return platform_driver_probe(&omap2_mcspi_driver, omap2_mcspi_probe);
    }
```

可见，为了 SPI 总线传输单独创建了 work queue，以保证整体的性能。

2. 总线传输接口

对于 SPI 总线的传输，由于性能的需求，主要使用 DMA 进行相关的操作。而在 SPI 总线框架，主要提供了两个传输接口，分别是 spi_master 中的 setup 和 transfer。下面看看 DM 3730 的具体实现过程。

先来看看 setup 接口，setup 接口主要是创建传输需要的资源，如申请 DMA 等。

```
    static int omap2_mcspi_setup(struct spi_device *spi)
    {
        int                         ret;
        struct omap2_mcspi          *mcspi;
        struct omap2_mcspi_dma          *mcspi_dma;
        struct omap2_mcspi_cs   *cs = spi->controller_state;
```

```
//检查字长是否是硬件支持的
if(spi->bits_per_word <4 || spi->bits_per_word >32){
    dev_dbg(&spi->dev, "setup: unsupported %d bit words\n",
            spi->bits_per_word);
    return -EINVAL;
}
//获得设备的控制器管理实体
mcspi = spi_master_get_devdata(spi->master);
mcspi_dma = &mcspi->dma_channels[spi->chip_select];

//如果没有为设备创建控制器相关的设备状态信息
if(!cs){
    cs = kzalloc(sizeof *cs, GFP_KERNEL);
    if(!cs)
        return -ENOMEM;
    //记录对应 CS 的寄存器地址
    cs->base = mcspi->base + spi->chip_select * 0x14;
    cs->phys = mcspi->phys + spi->chip_select * 0x14;
    //初始 channel 控制寄存器的值,单独保存是为了操作方便并且与电源管理相关
    cs->chconf0 =0;
    spi->controller_state = cs;
    /* Link this to context save list */
    //加入驱动中,进行电源管理相关的操作
    list_add_tail(&cs->node,&omap2_mcspi_ctx[mcspi->master->bus_num - 1].cs);
}

//如果没有则申请 DMA
if(mcspi_dma->dma_rx_channel == -1
                || mcspi_dma->dma_tx_channel == -1){
    /* TI81XX has EDMA and not SDMA, hence overriding SDMA usage */
    if(!cpu_is_ti81xx())
        ret = omap2_mcspi_request_dma(spi);
    else
        ret =0;
    if(ret <0)
        return ret;
}

//对控制器进行操作先要获得 clock
if(omap2_mcspi_enable_clocks(mcspi))
    return -ENODEV;
//下面要根据信号属性进行基本的寄存器设置
```

```
        ret = omap2_mcspi_setup_transfer(spi, NULL);
        omap2_mcspi_disable_clocks(mcspi);

        return ret;
}
```

这里主要是对于与控制器相关的设备状态的初始化、DMA 的申请，以及根据相应信号属性来初始化相关寄存器。

接下来看看 transfer 接口的实现过程：

```
static int omap2_mcspi_transfer(struct spi_device *spi, struct spi_message *m)
{
        struct omap2_mcspi          *mcspi;
        unsigned long               flags;
        struct spi_transfer     *t;
        //初始化传输状态
        m->actual_length = 0;
        m->status = 0;

        /* reject invalid messages and transfers */
        //链表是空或者没有设置回调是无效的传输,返回错误
        if(list_empty(&m->transfers) || !m->complete)
                return -EINVAL;
        //对每个 transfer 的属性进行检查和设置
        list_for_each_entry(t, &m->transfers, transfer_list){
                const void          *tx_buf = t->tx_buf;
                void                *rx_buf = t->rx_buf;
                unsigned    len = t->len;
                //检查 transfer 参数是否支持
                if(t->speed_hz > OMAP2_MCSPI_MAX_FREQ
                                || (len &&!(rx_buf || tx_buf))
                                || (t->bits_per_word &&
                        (   t->bits_per_word < 4 ||  t->bits_per_word > 32))){
                        dev_dbg(&spi->dev, "transfer: %d Hz, %d %s%s, %d bpw\n",
                                        t->speed_hz, len, tx_buf ? "tx" : "",
                                        rx_buf ? "rx" : "", t->bits_per_word);
                        return -EINVAL;
                }
                if(t->speed_hz && t->speed_hz < OMAP2_MCSPI_MAX_FREQ/(1 << 16)){
                        dev_dbg(&spi->dev, "%d Hz max exceeds %d\n",
                                        t->speed_hz,
                                        OMAP2_MCSPI_MAX_FREQ/(1 << 16));
```

```
                return -EINVAL;
            }

            if(m->is_dma_mapped || len < DMA_MIN_BYTES)
                continue;
            //使用流式 DMA 映射来分配 DMA 操作的空间
            if(tx_buf! = NULL){
                t->tx_dma = dma_map_single(&spi->dev,(void *) tx_buf,
                                len, DMA_TO_DEVICE);
                if(dma_mapping_error(&spi->dev, t->tx_dma)){
                    dev_dbg(&spi->dev, "dma %cX %d bytes error\n",
                                'T' , len);
                    return -EINVAL;
                }
            }
            if(rx_buf! = NULL){
                t->rx_dma = dma_map_single(&spi->dev, rx_buf,
                            t->len, DMA_FROM_DEVICE);
                if(dma_mapping_error(&spi->dev, t->rx_dma)){
                    dev_dbg(&spi->dev, "dma %cX %d bytes error\n",
                                'R' , len);
                    if(tx_buf! = NULL)
                        dma_unmap_single(NULL, t->tx_dma,
                                    len, DMA_TO_DEVICE);
                    return -EINVAL;
                }
            }
        }
        //获得控制器管理实体
        mcspi = spi_master_get_devdata(spi->master);
        //获得控制器的锁
        spin_lock_irqsave(&mcspi->lock, flags);
        //将传输信息加入驱动的队列,并唤醒 Work Queue 执行相应的 work
        list_add_tail(&m->queue, &mcspi->msg_queue);
        queue_work(omap2_mcspi_wq, &mcspi->work);
        spin_unlock_irqrestore(&mcspi->lock, flags);

        return 0;
    }
```

主要的工作仍在设置上，这里是为 DMA 分配空间建立映射，并唤醒 work queue 执行实际的操作。work 在初始化时已经设置了相应的操作 omap2_mcspi_work，这是执行实际的总

线传输的操作。下面看看实现细节：

```
static void omap2_mcspi_work(struct work_struct *work)
{
    struct omap2_mcspi          *mcspi;
    //找到控制器管理信息
    mcspi = container_of(work, struct omap2_mcspi, work);
    //操作中要获得锁
    spin_lock_irq(&mcspi->lock);
    //使能控制器时钟
    if(omap2_mcspi_enable_clocks(mcspi))
            goto out;
    //遍历控制器的操作链表,进行所有的操作
    while(!list_empty(&mcspi->msg_queue)) {
        struct spi_message              *m;
        struct spi_device                *spi;
        struct spi_transfer             *t = NULL;
        int                              cs_active = 0;
        struct omap2_mcspi_cs            *cs;
        struct omap2_mcspi_device_config *cd;
        int                              par_override = 0;
        int                              status = 0;
        u32                              chconf;
        //获得一个总线传输信息
        m = container_of(mcspi->msg_queue.next, struct spi_message,
                        queue);
        //将其移出队列
        list_del_init(&m->queue);
        //为避免长时间占用锁,这里释放
        spin_unlock_irq(&mcspi->lock);

        spi = m->spi;
        cs = spi->controller_state;
        cd = spi->controller_data;
        //总线控制器设置 channel 为 enable
        omap2_mcspi_set_enable(spi, 1);
        //对所有的 transfer 进行操作
        list_for_each_entry(t, &m->transfers, transfer_list) {
            if(t->tx_buf == NULL && t->rx_buf == NULL && t->len) {
                status = -EINVAL;
                break;
            }
```

```
if(par_override || t->speed_hz || t->bits_per_word){
    par_override = 1;
    status = omap2_mcspi_setup_transfer(spi, t);
    if(status < 0)
        break;
    if(!t->speed_hz && !t->bits_per_word)
        par_override = 0;
}
//先要设置 CS 信号
if(!cs_active){
    omap2_mcspi_force_cs(spi, 1);
    cs_active = 1;
}

//内存缓存了 channel 控制寄存器的值
chconf = mcspi_cached_chconf0(spi);
chconf &= ~OMAP2_MCSPI_CHCONF_TRM_MASK;
chconf &= ~OMAP2_MCSPI_CHCONF_TURBO;
//根据传输特性进行寄存器设置
if(t->tx_buf == NULL)
    chconf |= OMAP2_MCSPI_CHCONF_TRM_RX_ONLY;
else if(t->rx_buf == NULL)
    chconf |= OMAP2_MCSPI_CHCONF_TRM_TX_ONLY;

if(cd && cd->turbo_mode && t->tx_buf == NULL){
    /* Turbo mode is for more than one word */
    if(t->len > ((cs->word_len + 7) >> 3))
        chconf |= OMAP2_MCSPI_CHCONF_TURBO;
}
//写 channel 控制寄存器
mcspi_write_chconf0(spi, chconf);

if(t->len){
    unsigned count;

    /* RX_ONLY mode needs dummy data in TX reg */
    if(t->tx_buf == NULL)
        __raw_writel(0, cs->base + OMAP2_MCSPI_TX0);
    //根据需要进行 DMA 传输或者处理器直接进行操作
    if(m->is_dma_mapped || t->len >= DMA_MIN_BYTES)
        count = omap2_mcspi_txrx_dma(spi, t);
    else
```

```
                count = omap2_mcspi_txrx_pio( spi, t) ;
            //对操作数目状态进行更新
            m -> actual_length  + = count;

            if( count! = t -> len) {
                status = - EIO;
                break;
            }
        }
        //一次 transfer 结束,如果需要等待则等待一定时间
        if( t -> delay_usecs)
            udelay( t -> delay_usecs) ;

        /* ignore the "leave it on after last xfer" hint */
        //如果必要进行 CS 操作
        if( t -> cs_change) {
            omap2_mcspi_force_cs( spi, 0) ;
            //记录 CS 信号状态
            cs_active = 0;
        }
    }

    /* Restore defaults if they were overriden */
    if( par_override) {
        par_override = 0;
        status = omap2_mcspi_setup_transfer( spi, NULL) ;
    }
    if( cs_active)
        omap2_mcspi_force_cs( spi, 0) ;
    //总线控制器设置 channel 为 disable
    omap2_mcspi_set_enable( spi, 0) ;

    //传输结束,设置状态并完成总线框架层的同步操作.
    m -> status = status;
    m -> complete( m -> context) ;

    spin_lock_irq( &mcspi -> lock) ;
}

//关闭总线控制器相关 clock
omap2_mcspi_disable_clocks( mcspi) ;
```

```
out:
    spin_unlock_irq(&mcspi->lock);
}
```

可见 DM 3730 总线控制器的传输操作考虑了性能以及功耗的需求，通过单独的内核执行实体进行操作，并考虑到效率的需求，根据传输数据量进行 DMA 或者处理器直接操作的方式进行实际的 IO 操作，实际的操作参见相关代码。

到这里整个传输从设置到实际操作就介绍完了。

7.2.4 SPI 总线驱动电源管理相关说明

关于电源管理，先看看 SPI 总线层面提供的功能：

```
struct bus_type spi_bus_type = {
    .name           = "spi",
    .dev_attrs      = spi_dev_attrs,
    .match          = spi_match_device,
    .uevent         = spi_uevent,
    .suspend = spi_suspend,
    .resume         = spi_resume,
};
```

可见只提供了 SLM 的相关功能，下面以 suspend 为例看看具体的实现过程：

```
static int spi_suspend(struct device *dev, pm_message_t message)
{
    int                 value = 0;
    struct spi_driver   *drv = to_spi_driver(dev->driver);

    /* suspend will stop irqs and dma; no more i/o */
    if(drv){
        if(drv->suspend)
            value = drv->suspend(to_spi_device(dev), message);
        else
            dev_dbg(dev, "... can't suspend\n");
    }
    return value;
}
```

从代码中可见，主要的操作是通过总线设备的驱动来完成的，其中通常为关闭资源的操作。

总线层面是针对总线设备进行的操作，再来看看总线控制器相关的电源管理功能。DM 3730 SPI 总线控制器的 runtime 电源管理功能集中在传输中的操作，由前面分析的代码中可

知，主要通过 omap2_mcspi_disable_clocks、omap2_mcspi_enable_clocks 和 omap2_mcspi_set_enable 来实现模块时钟、模块开关等电源管理相关的操作。其他的电源管理功能则由 omap2_mcspi_pm_ops 来提供，具体内容如下：

```
static const struct dev_pm_ops omap2_mcspi_pm_ops = {
      . resume = omap2_mcspi_resume,
};
```

为什么没有 suspend 操作呢？相应的 suspend 操作是在 platform bus 的 suspend 接口中实现的，SPI 总线控制器也是使用统一的操作接口。在从低功耗状态进行恢复的时候 SPI 总线控制器会将 CS 置为 active 状态，所以这里需要特殊的操作。下面来看看具体实现过程：

```
static int omap2_mcspi_resume(struct device  *dev)
{
      struct spi_master        *master = dev_get_drvdata(dev);
      struct omap2_mcspi            *mcspi = spi_master_get_devdata(master);
      struct omap2_mcspi_cs  *cs;
      //进行操作需要使能 clock
      omap2_mcspi_enable_clocks(mcspi);
      //对所有连接外设的 channel 控制器进行设置
      list_for_each_entry(cs, &omap2_mcspi_ctx[master->bus_num - 1].cs,
                        node) {
            //检查之前的 CS 状态是否为 deactive
            if((cs->chconf0 & OMAP2_MCSPI_CHCONF_FORCE) == 0) {
                  //进行设置,保证为 deactive 状态
                  MOD_REG_BIT(cs->chconf0, OMAP2_MCSPI_CHCONF_FORCE, 1);
                  __raw_writel(cs->chconf0, cs->base + OMAP2_MCSPI_CHCONF0);
                  MOD_REG_BIT(cs->chconf0, OMAP2_MCSPI_CHCONF_FORCE, 0);
                  __raw_writel(cs->chconf0, cs->base + OMAP2_MCSPI_CHCONF0);
            }
      }
      //关闭相关时钟
      omap2_mcspi_disable_clocks(mcspi);
      return 0;
}
```

可见电源管理相关的操作也会涉及功能部分。这样电源管理部分的功能就完整了。

7.3 多媒体卡（MMC）

7.3.1 MMC 需求

MMC（即 multimedia card）已经成为嵌入式存储设备的最重要的连接标准。它于 1997

年由西门子与 SanDisk 共同开发，技术基于 NAND 技术，随后又有不同的发展，现如今已经形成 MMC、SD、eMMC、SDIO 等不同的形式，而连接的设备也是多种多样的，从 MMC Card、SD Card、microSD Card 等存储设备，到 SDIO WLAN、SDIO GPS 等设备。总之设备形态以存储设备为主，相应的 SDIO 总线可以支持不同功能的设备。

MMC 的架构如图 7-8 所示。图 7-8 引自《MMC 规范》。

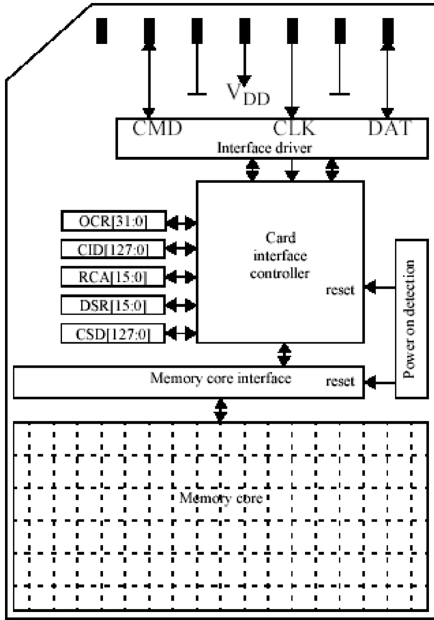


图 7-8　MMC 架构图

从图 7-8 可见，MMC 连接除了 CLK 和 VDD 之外还有 CMD 和 DAT 信号，其中 CMD 信号用于命令交互，而 DAT 信号用于数据交互，DAT 信号可以有多个。MMC 内部会有一些寄存器，这些寄存器记录了 MMC 的能力等属性，可用于与总线控制器协商。需要根据这些控制器的属性值正确地操作 MMC 设备。

MMC 的信号传输特点如图 7-9 所示。图 7-9 引自《DM 3730 芯片手册》中第 3384 页框图。

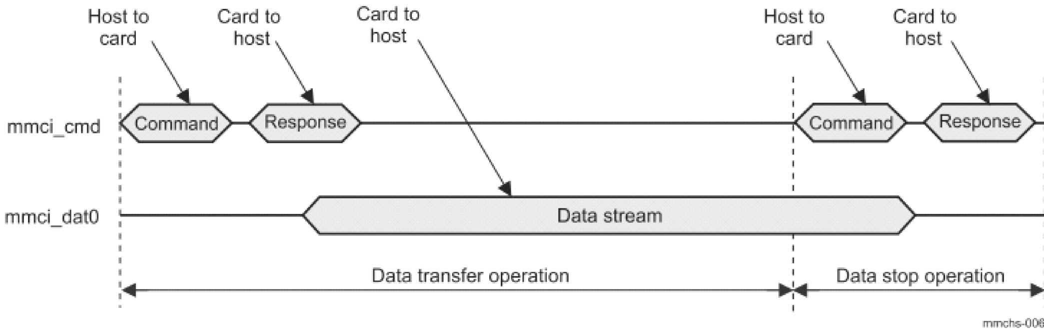


图 7-9　MMC 信号传输特点

图 7-9 以单一的读操作为例，可见 MMC 的信号主要通过 CMD 传送命令，并且需要响应，通过 DAT 信号传送数据。为了能正确地识别 MMC 设备，并正确地传输数据，MMC 规范规定了不同的模式及命令用于不同的功能，通过命令可以在不同的状态转换。这些都属于规范中状态及状态转换的内容，能正确地支持。

MMC 总线可以连接不同协议标准的设备，如 MMC、SD、SDIO，这些都需要在框架中进行支持。另外由于 MMC 连接的设备可能会以卡的形式存在，这就需要能够支持对设备插入的检测，在系统级别也需要能够对设备插拔进行支持。

总体上来说，MMC 总线驱动的需求就是要能实现总线的各种功能，并且满足总线的各种无关性需求。

7.3.2 MMC 框架解析

1. 总线相关以及核心框架

MMC 总线设备相对于 I^2C 和 SPI 总线要复杂，但是总线框架的思路是一致的。首先来看看 MMC 子系统的初始化 mmc_init。

```
static int __init mmc_init(void)
{
	int ret;
	//创建 work queue
	workqueue = create_singlethread_workqueue("kmmcd");
	if(!workqueue)
		return -ENOMEM;
	//注册 mmc bus
	ret = mmc_register_bus();
	if(ret)
		goto destroy_workqueue;
	//注册 mmc host class
	ret = mmc_register_host_class();
	if(ret)
		goto unregister_bus;
	//注册 sdio bus
	ret = sdio_register_bus();
	if(ret)
		goto unregister_host_class;

	return 0;
...
}
```

从分析可见，注册了两个 bus（分别是 mmc bus 和 sdio bus）和一个 class（mmc host class），这里有两种类型的 bus 是由于 MMC 连接还需要支持 SDIO 总线类型，而在 MMC 框架中就需要实现 SDIO 总线的功能，其中 SDIO 总线建立在 MMC 总线的基础上，所以需要注

册两种 bus。而注册一个 class 的原因，与 SPI 设备管理部分的实现是相同的，这样是为了避免总线控制器的设备进行总线相关的操作。

接下来看看两个总线的信息：

```
static struct bus_type mmc_bus_type = {
    .name        = "mmc",
    .dev_attrs   = mmc_dev_attrs,
    .match       = mmc_bus_match,
    .uevent      = mmc_bus_uevent,
    .probe       = mmc_bus_probe,
    .remove      = mmc_bus_remove,
    .suspend = mmc_bus_suspend,
    .resume      = mmc_bus_resume,
    .pm      = MMC_PM_OPS_PTR,
};

static struct bus_type sdio_bus_type = {
    .name        = "sdio",
    .dev_attrs   = sdio_dev_attrs,
    .match       = sdio_bus_match,
    .uevent      = sdio_bus_uevent,
    .probe       = sdio_bus_probe,
    .remove      = sdio_bus_remove,
    .pm      = SDIO_PM_OPS_PTR,
};
```

关于这两种 bus 先来了解 MMC 总线的情况：

```
static int mmc_bus_match(struct device *dev, struct device_driver *drv)
{
    return 1;
}
```

match 直接返回真，所以只要是 MMC 总线设备就会直接与 MMC 总线驱动匹配，这样的情况说明 MMC 总线驱动只有一个，并由系统框架已经提供，可以支持所有的设备，这与 MMC 总线设备主要作为存储设备相关，存储设备只要提供统一的块设备驱动（mmc block 层的驱动）即可。

现在对 MMC 总线管理的设备还是不清楚，继续看一下 probe 接口：

```
static int mmc_bus_probe(struct device *dev)
{
    struct mmc_driver *drv = to_mmc_driver(dev->driver);
    struct mmc_card *card = mmc_dev_to_card(dev);
```

```
        return drv -> probe(card);
    }
```

从中可见，MMC 总线设备主要就是 mmc_card，而总线驱动就是 mmc_driver，在 mmc_driver 的 probe 接口中会对上层的驱动相关的接口进行初始化，建立上层与 MMC 总线层的关联，主要就是与块设备驱动层的关联。

这样的设计与实现是合理的，因为 MMC 的设备（包括 SD 卡设备）主要用于数据存储，所以块设备是最合适的。

而 SDIO 则不同，SDIO 设备可以借助 MMC 总线连接实现不同的功能，下面看看 SDIO 总线的相关接口：

```
static int sdio_bus_match(struct device *dev, struct device_driver *drv)
{
    struct sdio_func *func = dev_to_sdio_func(dev);
    struct sdio_driver *sdrv = to_sdio_driver(drv);

    if(sdio_match_device(func, sdrv))
        return 1;

    return 0;
}
```

从 match 中可见，会涉及具体的总线设备与总线驱动的匹配，这是因为 SDIO 可以有不同的功能，所以需要进行匹配。

SDIO 总线管理的设备在内核中主要由 sdio_func 来表示，而相对应的驱动则由 sdio_driver 来表示。

MMC 整体框架如图 7-10 所示。

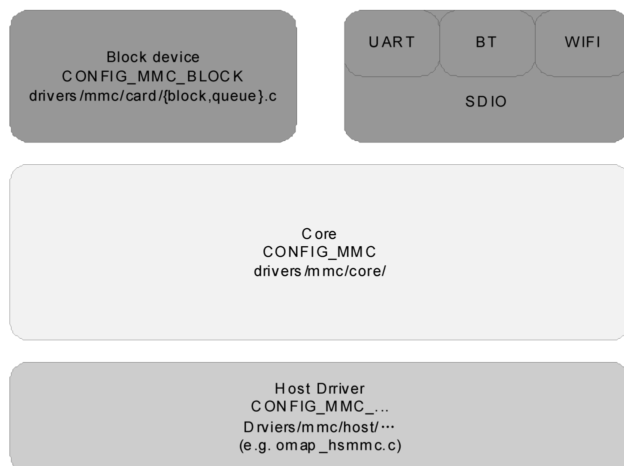


图 7-10　MMC 整体框架

从图 7-10 可见，MMC 主要管理两个总线功能：MMC 和 SDIO。MMC 总线功能对应于块设备，而在 SDIO 总线之上可以是具体功能的设备。通常一个 MMC 接口只与一个设备进行连接，但是因为允许插拔，所以需要能支持检测、扫描来支持硬件插拔，以及不同功能设备的切换。

MMC 总线整体框架主要实现了这些功能。

2. 总线控制器相关

在 MMC 框架中，无论是 MMC 总线还是 SDIO 总线，实际的总线控制器都是相同的硬件，只是协议中命令字和状态机的差别。在介绍 MMC 框架初始代码时见到其中注册了 class，以与 MMC 总线设备进行区分，相应的 class 就是为总线控制器提供的，MMC 框架中总线控制器由 mmc_host 来表示和管理，细节如下：

```
struct mmc_host {
    //设备层次关系通常是 platform device 中的 device 实体
    struct device           *parent;
    //设备模型对应的设备,属于 mmc_host_class
    struct device           class_dev;
    //编号
    int                     index;
    //设备相关的控制器操作接口
    const struct mmc_host_ops *ops;
    //总线频率相关属性
    unsigned int            f_min;
    unsigned int            f_max;
    unsigned int            f_init;
    //支持的操作状态,主要是允许的电压范围
    u32                     ocr_avail;
    //接受电源管理的通知操作接口
    struct notifier_block   pm_notify;
    //控制器的能力,主要是接口能力的设置,如 4/8bit 传输,速度能力等
    unsigned long           caps;           /* Host capabilities */
    //电源管理的能力
    mmc_pm_flag_t           pm_caps;    /* supported pm features */
    /* host specific block data */
    //块设备相关的能力信息,如最大块大小,一个请求的限制,与控制器的能力相关
    unsigned int            max_seg_size;   /* see blk_queue_max_segment_size */
    unsigned short          max_segs;       /* see blk_queue_max_segments */
    unsigned short          unused;
    unsigned int            max_req_size;   /* maximum number of bytes in one req */
    unsigned int            max_blk_size;   /* maximum size of one mmc block */
    unsigned int            max_blk_count;  /* maximum number of blocks in one req */

    /* private data */
```

```
    spinlock_t              lock;           /* lock for claim and bus ops */
    //当前 MMC IO 接口的状态
    struct mmc_ios              ios;        /* current io bus settings */
    //当前操作状态的设置
    u32                 ocr;        /* the current OCR setting */
    /* group bitfields together to minimize padding */
    unsigned int            use_spi_crc:1;
    //控制器的占用状态
    unsigned int            claimed:1;      /* host exclusively claimed */
    unsigned int            bus_dead:1;     /* bus has been released */

    /* Only used with MMC_CAP_DISABLE */
    //当控制器允许 disable 会有如下属性
    int                 enabled;  /* host is enabled */
    int                 rescan_disable;     /* disable card detection */
    int                 nesting_cnt;    /* "enable" nesting count */
    int                 en_dis_recurs; /* detect recursion */
    unsigned int            disable_delay; /* disable delay in msecs */
    struct delayed_work         disable;  /* disabling work */

    //连接的 MMC 设备
    struct mmc_card             *card;          /* device attached to this host */
    //等待操作的任务
    wait_queue_head_t wq;
    //当前对控制器进行控制的任务
    struct task_struct      *claimer; /* task that has host claimed */
    int                 claim_cnt;      /* "claim" nesting count */

    //进行 detect 检查设备的 work
    struct delayed_work         detect;

    //当前所属的总线操作
    const struct mmc_bus_ops *bus_ops;      /* current bus driver */
    unsigned int            bus_refs; /* reference counter */

    //SDIO 的相关中断
    unsigned int            sdio_irqs;
    struct task_struct      *sdio_irq_thread;
    atomic_t            sdio_irq_thread_abort;
    //请求的 pm 状态,通常与 SDIO card 相关
    mmc_pm_flag_t               pm_flags;       /* requested pm features */
...
};
```

可见主要是对总线控制器的操作能力以及状态进行设置，这些操作能力相关的都是属于相关规范的物理特性，需要通过这些属性标注总线控制器物理能力。另外重要的就是两个操作接口，分别是 mmc_host_ops 和 mmc_bus_ops。

先来看看 mmc_host_ops：

```
struct mmc_host_ops {
    //电源管理相关的 enable 和 disable 操作接口
    int( * enable)(struct mmc_host  * host);
    int( * disable)(struct mmc_host  * host, int lazy);

    //传输的操作接口
    void( * request)(struct mmc_host  * host, struct mmc_request  * req);

    //IO 状态的设置接口
    void( * set_ios)(struct mmc_host  * host, struct mmc_ios  * ios);
    //获得只读状态
    int   ( * get_ro)(struct mmc_host  * host);
    //获得当前 card 是否在位的状态
    int   ( * get_cd)(struct mmc_host  * host);
    //对 SDIO 接口中断 enable 和 disable 的操作接口
    void( * enable_sdio_irq)(struct mmc_host  * host, int enable);
    //可选的对设备初始化的接口
    void( * init_card)(struct mmc_host  * host, struct mmc_card  * card);
};
```

这里主要是控制和进行传输操作的接口，是进行实际总线控制器操作的接口，真正的传输也是通过总线控制器来完成的。

mmc_bus_ops 接口细节如下：

```
struct mmc_bus_ops {
    int( * awake)(struct mmc_host  * );
    int( * sleep)(struct mmc_host  * );
    void( * remove)(struct mmc_host  * );
    void( * detect)(struct mmc_host  * );
    int( * suspend)(struct mmc_host  * );
    int( * resume)(struct mmc_host  * );
    int( * power_save)(struct mmc_host  * );
    int( * power_restore)(struct mmc_host  * );
};
```

需要 mmc_bus_ops 的原因是因为 MMC 总线可以支持不同协议规范的设备，这些设备相关的操作是不同的，所以在控制器中提供相应的操作接口，这样在总线设备侦测时就可以根据设备的特点加载合适的操作接口。在相应的操作中，detect 是检测设备是否存在，其他大

多与电源管理相关，通过规范的命令可以完成相应的操作。

MMC 总线控制器框架提供了创建的接口 mmc_alloc_host，主要内容如下：

```
struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
{
    int err;
    struct mmc_host *host;

    if(!idr_pre_get(&mmc_host_idr, GFP_KERNEL))
        return NULL;
    //分配空间,这里会为总线控制器设备相关的管理实体分配空间大小为 extra
    host = kzalloc(sizeof(struct mmc_host) + extra, GFP_KERNEL);
    if(!host)
        return NULL;

    spin_lock(&mmc_host_lock);
    //获得新的编号,建立 ID 号和管理实体指针间的关联
    err = idr_get_new(&mmc_host_idr, host, &host->index);
    spin_unlock(&mmc_host_lock);
    if(err)
        goto free;
    //设置设备模型相关的名字
    dev_set_name(&host->class_dev, "mmc%d", host->index);
    //设备模型相关初始化
    host->parent = dev;
    host->class_dev.parent = dev;
    host->class_dev.class = &mmc_host_class;
    device_initialize(&host->class_dev);

    spin_lock_init(&host->lock);
    //初始化不同任务操作控制器的等待队列
    init_waitqueue_head(&host->wq);
    //初始化探测设备的 work
    INIT_DELAYED_WORK(&host->detect, mmc_rescan);
    //初始化电源管理相关的 work
  INIT_DELAYED_WORK_DEFERRABLE(&host->disable,mmc_host_deeper_disable);
#ifdef CONFIG_PM
    //电源管理相关的通知操作接口
    host->pm_notify.notifier_call = mmc_pm_notify;
#endif

    //设置操作的能力
```

```
        host -> max_segs = 1;
        host -> max_seg_size = PAGE_CACHE_SIZE;

        host -> max_req_size = PAGE_CACHE_SIZE;
        host -> max_blk_size = 512;
        host -> max_blk_count = PAGE_CACHE_SIZE / 512;

        return host;
...
}
```

主要的操作是分配空间，并初始化与功能相关的 work，其中最重要的是探测设备的 work，相应的操作是 mmc_rescan，后面会进行介绍。

接下来的接口是注册并初始化控制器硬件的接口 mmc_add_host，详细内容如下：

```
int mmc_add_host(struct mmc_host *host)
{
        int err;
...
        //注册设备模型
        err = device_add(&host -> class_dev);
        if(err)
                return err;
...
        //启动总线控制器
        mmc_start_host(host);
        //注册相应电源管理通知的操作
        if(!(host -> pm_flags & MMC_PM_IGNORE_PM_NOTIFY))
                register_pm_notifier(&host -> pm_notify);

        return 0;
}
```

在调用 mmc_add_host 之前，需要将 mmc_host_ops 初始化，并将控制器的相关属性进行正确设置。其中最重要的操作就是 mmc_start_host，会通过 mmc_detect_change 唤醒控制器的 detect work，最终会通过 mmc_rescan 来进行设备探测，这实现了 MMC 总线设备发现的重要功能，详细分析如下：

```
void mmc_rescan(struct work_struct *work)
{
        //先获得总线控制器实体
        struct mmc_host *host =
```

```
        container_of(work, struct mmc_host, detect.work);
    u32 ocr;
    int err;
    unsigned long flags;
    int i;
    const unsigned freqs[] = { 400000, 300000, 200000, 100000 };
    int extend_wakelock = 0;

    spin_lock_irqsave(&host->lock, flags);
    //如果不允许探测则返回
    if(host->rescan_disable){
        spin_unlock_irqrestore(&host->lock, flags);
        return;
    }

    spin_unlock_irqrestore(&host->lock, flags);
    //计数并使能控制器
    mmc_bus_get(host);
    //如果设备可以移除,检查设备是否存在
    if(host->bus_ops && host->bus_ops->detect &&!host->bus_dead
        && mmc_card_is_removable(host))
        host->bus_ops->detect(host);
…
    mmc_bus_put(host);
    mmc_bus_get(host);
    /* if there still is a card present, stop here */
    //detect 操作已经发现合适设备则退出
    if(host->bus_ops!= NULL){
        mmc_bus_put(host);
        goto out;
    }
    mmc_bus_put(host);
    //检测到没有设备存在
    if(host->ops->get_cd && host->ops->get_cd(host) == 0)
        goto out;
    //按照总线频率从高到低探测设备
    for(i = 0; i < ARRAY_SIZE(freqs); i++){
        //声明占用控制器
        mmc_claim_host(host);
        //设置总线频率值
        if(freqs[i] >= host->f_min)
            host->f_init = freqs[i];
```

```
        else if( ! i || freqs[ i - 1 ] > host -> f_min)
            host -> f_init = host -> f_min;
        else {
            mmc_release_host( host) ;
            goto out;
        }
...
        mmc_power_up( host) ;
        sdio_reset( host) ;
        mmc_go_idle( host) ;

        mmc_send_if_cond( host, host -> ocr_avail) ;

        //首先检测是否是 SDIO
        err = mmc_send_io_op_cond( host, 0, &ocr) ;
        if( ! err) {
            //特殊命令操作成功则绑定 SDIO 相关 bus ops,以及对 SDIO 设备
            //进行检测初始化
            if( mmc_attach_sdio( host, ocr) ) {
                mmc_claim_host( host) ;
                //检测不成功则检查是否是 SD 设备
                if( mmc_send_app_op_cond( host, 0, &ocr) )
                    goto out_fail;
                //特殊命令操作成功则绑定 SD 相关 bus ops,以及对 SD 设备
                //进行检测初始化
                if( mmc_attach_sd( host, ocr) )
                    mmc_power_off( host) ;
                extend_wakelock = 1;
            }
            goto out;
        }
        //检测是否是 SD 设备
        err = mmc_send_app_op_cond( host, 0, &ocr) ;
        if( ! err) {
            //特殊命令操作成功则绑定 SD 相关 bus ops,以及对 SD 设备
            //进行检测初始化
            if( mmc_attach_sd( host, ocr) )
                mmc_power_off( host) ;
            extend_wakelock = 1;
            goto out;
        }
        //检测是否是 MMC 设备
```

```
            err = mmc_send_op_cond(host, 0, &ocr);
            if(!err){
                //特殊命令操作成功则绑定 MMC 相关 bus ops,以及对 MMC 设备
                //进行检测初始化
                if(mmc_attach_mmc(host, ocr))
                    mmc_power_off(host);
                extend_wakelock = 1;
                goto out;
            }

out_fail:
            mmc_release_host(host);
            mmc_power_off(host);
        }
...
}
```

可见主要的工作就是探测设备，并在探测成功后通过 mmc_attach_xxx 来初始化总线控制器的 bus ops 接口，并对总线设备进行初始化操作。

这样总线的主要功能，包括总线传输事务以及总线设备发现，就都在代表 MMC 总线的 MMC 总线控制器部分有了完整支持。

3. 总线设备相关

前面已经提及，MMC 总线设备管理由 mmc_card 来完成。下面具体介绍：

```
struct mmc_card {
    //关联的总线控制器
    struct mmc_host          *host;   /* the host this device belongs to */
    //设备模型实体
    struct device        dev;        /* the device */
    //SD 需要的相对 card 地址
    unsigned int         rca;        /* relative card address of device */
    //card 的类型主要是 MMC、SD、SDIO 或者 SDIO/SD combo 的
    unsigned int         type;             /* card type */
    //card 的状态,如是否只读、速度状态等
    unsigned int         state;            /* (our) card state */
    //特殊属性
    unsigned int         quirks;    /* card quirks */
    //擦除相关的属性,由于基于 nand 的存储架构,所有需要擦除相关的属性
    unsigned int         erase_size;      /* erase size in sectors */
    unsigned int         erase_shift;     /* if erase unit is power 2 */
    unsigned int         pref_erase;      /* in sectors */
    u8                   erased_byte;     /* value of erased bytes */
```

```
    //card 上相关寄存器的 raw 值
    u32                 raw_cid[4];     /* raw card CID */
    u32                 raw_csd[4];     /* raw card CSD */
    u32                 raw_scr[2];     /* raw card SCR */
    //转换后的 MMC card 上相关属性
    struct mmc_cid          cid;        /* card identification */
    struct mmc_csd          csd;        /* card specific */
    struct mmc_ext_csd      ext_csd;    /* mmc v4 extended card specific */
    //转换后的 SD card 上相关属性
    struct sd_scr           scr;        /* extra SD information */
    struct sd_ssr           ssr;        /* yet more SD information */
    struct sd_switch_caps   sw_caps;    /* switch(CMD6) caps */

    //SDIO 一个 card 上可以支持多个功能,这里是功能数目
    unsigned int            sdio_funcs; /* number of SDIO functions */
    //转换后的 SDIO card 上相关属性
    struct sdio_cccr        cccr;       /* common card info */
    struct sdio_cis         cis;        /* common tuple info */
    //SDIO 每个功能的管理实体,每个功能在逻辑上都相当于一个设备
    struct sdio_func    *sdio_func[SDIO_MAX_FUNCS]; /* SDIO functions(devices) */
    //一些信息
    unsigned            num_info;       /* number of info strings */
    const char          **info;         /* info strings */
    struct sdio_func_tuple  *tuples;    /* unknown common tuples */
    struct dentry           *debugfs_root;
};
```

从中可见，为了支持 MMC/SD/SDIO 等不同的协议规范，其中的属性都在 mmc_card 中进行了表示，具体的类型和对应的属性都会在设备发现阶段进行初始化。

MMC 设备的管理框架提供了相应的创建和注册接口。首先来了解创建的接口 mmc_alloc_card：

```
struct mmc_card *mmc_alloc_card(struct mmc_host *host, struct device_type *type)
{
    struct mmc_card *card;
    //分配空间
    card = kzalloc(sizeof(struct mmc_card), GFP_KERNEL);
    if(!card)
        return ERR_PTR(-ENOMEM);
    //与总线控制器关联
    card->host = host;
    //设备模型信息的初始化
```

```
        device_initialize(&card->dev);
        //设备层次关联,总线设置
        card->dev.parent = mmc_classdev(host);
        card->dev.bus = &mmc_bus_type;
        card->dev.release = mmc_release_card;
        card->dev.type = type;

        return card;
}
```

主要进行设备模型相关的设置。下面的接口是用于注册的，详细内容如下：

```
int mmc_add_card(struct mmc_card *card)
{
        int ret;
        const char *type;
        //根据控制器名字和卡的相对 card 地址创建设备名字
        dev_set_name(&card->dev, "%s:%04x", mmc_hostname(card->host), card->rca);
...
        //进行设备模型注册
        ret = device_add(&card->dev);
        if(ret)
                return ret;
...
        //设置卡存在的状态
        mmc_card_set_present(card);

        return 0;
}
```

可见主要的工作与设备模型相关，当向设备模型注册了 device 之后，就会通过总线来进行驱动的 probe 操作。对 MMC 驱动，之前已经说明只有 MMC block 提供的 mmcblk 驱动，详细内容如下：

```
static struct mmc_driver mmc_driver = {
        .drv        = {
                .name       = "mmcblk",
        },
        .probe              = mmc_blk_probe,
        .remove             = mmc_blk_remove,
        .suspend = mmc_blk_suspend,
        .resume             = mmc_blk_resume,
};
```

主要与块设备层关联的工作是由 mmc_blk_probe 来完成的，下面来看看细节：

```
static int mmc_blk_probe(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int err;
    char cap_str[10];

    //检查设备是否支持块读取,驱动需要设备允许块读取
    if(!(card->csd.cmdclass & CCC_BLOCK_READ))
        return -ENODEV;
    //这里分配 MMC 驱动相关的块设备管理实体,其中会创建块设备层需要的
    //的 gendisk,并通过 blk_init_queue 初始化 request queue 操作
    md = mmc_blk_alloc(card);
    if(IS_ERR(md))
        return PTR_ERR(md);
    //设置操作大小 512B,与 Linux 块设备层一致
    err = mmc_blk_set_blksize(md, card);
    if(err)
      goto out;
…
    //为设备模型设置管理实体,以进行电源管理相关操作
    mmc_set_drvdata(card, md);
    //加入 Linux 块设备层
    add_disk(md->disk);
    return 0;
…
}
```

从分析中可见，通过 mmcblk 驱动就可以将 MMC 设备与块驱动层建立关联，进而可以被文件系统和应用使用。至于 request queue 的操作，在 MMC block 层 mmc_init_queue 中单独创建了 kthread 进行相应操作：

```
mq->thread = kthread_run(mmc_queue_thread, mq, "mmcqd/%d", host->index);
```

在 mmc_queue_thread 中则从 request queue 中获得 request 再通过 mmc_queue 的 issue_fn 接口即 mmc_blk_issue_rq 来真正执行相应的 request。这样 MMC 作为存储设备的功能就完整了。

SDIO 总线设备由 sdio_func 进行管理，其详细内容就不进行分析了，与 mmc_card 类似，包含一些属性，但是由于 SDIO 中的 data0 可以作为中断源，所以其中包含中断的属性，总线控制器与 mmc_card 没有差别，所以不需要单独关联。MMC 框架也提供了创建和注册 SDIO 设备的接口。先来看看创建的接口 sdio_alloc_func：

```
struct sdio_func *sdio_alloc_func(struct mmc_card *card)
{
    struct sdio_func *func;
    //分配空间
    func = kzalloc(sizeof(struct sdio_func), GFP_KERNEL);
    if(!func)
        return ERR_PTR(-ENOMEM);
    //与 MMC 设备关联
    func->card = card;
    //设备模型初始化
    device_initialize(&func->dev);
    //注意这里的设备层次和总线类型关系
    func->dev.parent = &card->dev;
    func->dev.bus = &sdio_bus_type;
    func->dev.release = sdio_release_func;

    return func;
}
```

从分析中可见，SDIO 设备并不是修改 mmc_card 中的 dev 属性，而是拥有单独设备模型实体，在设备层次中其父设备是 mmc_card，这与设备的逻辑层次相符，而 mmc_card 在 sdio_func 中起到了中介作用。

再来看看设备注册的接口 sdio_add_func：

```
int sdio_add_func(struct sdio_func *func)
{
    int ret;
    //根据 card 号以及 function 编号
    dev_set_name(&func->dev, "%s:%d", mmc_card_id(func->card), func->num);
    //向设备模型进行注册
    ret = device_add(&func->dev);
    if(ret == 0)
        sdio_func_set_present(func);

    return ret;
}
```

可见主要的操作也是向设备模型进行注册，注册后就通过总线操作查找匹配的驱动并绑定，就可实现其相关功能。

对不同规范类型的设备，MMC 框架通过 mmc_xxx_init_card 和 mmc_attach_xxx 等接口封装了以上的操作来实现设备的创建与注册，这些都是在设备发现阶段完成的。

在设备发现阶段 MMC 主要依靠探测发现的方式进行，当 card 有插拔的时候同样需要进

行设备的重新发现与探测，MMC 框架为这种情况提供了接口 mmc_detect_change，其中会唤醒总线控制器的 detect 操作即 mmc_rescan 来重新 detect 总线设备的情况，并根据当前设备情况进行相关操作。

4. 总线传输接口

MMC 框架是通过 mmc_request 管理 MMC 总线传输，细节如下：

```
struct mmc_request {
    //进行操作的命令
    struct mmc_command      * cmd;
    //相关的数据
    struct mmc_data             * data;
    //操作结束的命令
    struct mmc_command      * stop;

    //同步操作的参数
    void                    * done_data;      /* completion data */
    //同步操作的接口
    void                    ( * done)(struct mmc_request * );/* completion function */
};
```

在总线层所有的传输都是通过 mmc_request 来进行实际操作。具体的接口是 mmc_wait_for_req，下面来看看细节：

```
void mmc_wait_for_req(struct mmc_host * host, struct mmc_request * mrq)
{
    DECLARE_COMPLETION_ONSTACK(complete);
    //设置完成同步参数和操作接口
    mrq -> done_data = &complete;
    mrq -> done = mmc_wait_done;
    //开始操作
    mmc_start_request(host, mrq);
    //等待完成
    wait_for_completion(&complete);
}
```

可见主要是以同步的方式进行总线传输操作的，具体操作的发起是通过 mmc_start_request 来完成的，细节如下：

```
static void mmc_start_request(struct mmc_host * host, struct mmc_request * mrq)
{
…
    //控制器上灯的显示
    led_trigger_event(host -> led, LED_FULL);
```

```
        //初始化状态并与 req 进行
        mrq -> cmd -> error = 0;
        mrq -> cmd -> mrq = mrq;
        if( mrq -> data) {
            //如果有数据操作,不能超过控制器的范围
            BUG_ON( mrq -> data -> blksz > host -> max_blk_size);
            BUG_ON( mrq -> data -> blocks > host -> max_blk_count);
            BUG_ON( mrq -> data -> blocks * mrq -> data -> blksz >
                    host -> max_req_size);
    …
            //初始化数据部分状态,并与 req 进行关联
            mrq -> cmd -> data = mrq -> data;
            mrq -> data -> error = 0;
            mrq -> data -> mrq = mrq;
            if( mrq -> stop) {
                //如果需要 stop 命令,初始化状态信息并与 req 进行关联
                mrq -> data -> stop = mrq -> stop;
                mrq -> stop -> error = 0;
                mrq -> stop -> mrq = mrq;
            }
        }
        //通过总线控制器进行操作
        host -> ops -> request( host, mrq);
    }
```

总线传输基本操作是先发送命令，然后进行数据操作，接下来结束命令。实际操作中后两步可选，在完整的操作中任何一步有了问题都会导致整个操作失败，相应的 mmc_request 中同样分为三步，并在三步的实体中都关联到 mmc_request 来保证正确处理。块设备的传输最终直接调用 mmc_wait_for_req 来完成。

除了数据等稍微复杂的传输操作之外，MMC 框架对简单的命令发送还提供了 mmc_wait_for_cmd 接口来执行相关操作。细节如下：

```
    int mmc_wait_for_cmd( struct mmc_host * host, struct mmc_command * cmd, int retries)
    {
        struct mmc_request mrq;

        WARN_ON( ! host -> claimed);
        //初始化 mmc_request
        memset( &mrq, 0, sizeof( struct mmc_request) );
        //初始化命令的响应数据,响应 respond;规范中规定不同命令的不同响应格式
        memset( cmd -> resp, 0, sizeof( cmd -> resp) );
        //重试次数
```

```
        cmd -> retries = retries;
        //命令
        mrq.cmd = cmd;
        cmd -> data = NULL;
        //等待同步完成
        mmc_wait_for_req(host, &mrq);

        return cmd -> error;
    }
```

可见主要的操作就是对 mmc_wait_for_req 的简单封装，其中主要是为了命令的操作。

MMC/SD/SDIO 规范中都规定了一些特殊的操作和命令，虽然这些操作最终都是通过以上的接口执行的，但是为了方便开发 MMC 框架分别提供了相应的接口进行相关操作，具体的操作可以参考规范的说明。具体的接口如下：

```
    //MMC 相关的操作
    int mmc_send_op_cond(struct mmc_host *host, u32 ocr, u32 *rocr);
    int mmc_all_send_cid(struct mmc_host *host, u32 *cid);
    int mmc_set_relative_addr(struct mmc_card *card);
    int mmc_send_csd(struct mmc_card *card, u32 *csd);
    int mmc_send_ext_csd(struct mmc_card *card, u8 *ext_csd);
    int mmc_switch(struct mmc_card *card, u8 set, u8 index, u8 value);
    int mmc_send_status(struct mmc_card *card, u32 *status);
    int mmc_send_cid(struct mmc_host *host, u32 *cid);

    //SD 相关的操作
    int mmc_app_set_bus_width(struct mmc_card *card, int width);
    int mmc_send_app_op_cond(struct mmc_host *host, u32 ocr, u32 *rocr);
    int mmc_send_if_cond(struct mmc_host *host, u32 ocr);
    int mmc_send_relative_addr(struct mmc_host *host, unsigned int *rca);
    int mmc_app_send_scr(struct mmc_card *card, u32 *scr);
    int mmc_sd_switch(struct mmc_card *card, int mode, int group, u8 value, u8 *resp);
    int mmc_app_sd_status(struct mmc_card *card, void *ssr);

    //SDIO 相关的操作
    int mmc_send_io_op_cond(struct mmc_host *host, u32 ocr, u32 *rocr);
    int mmc_io_rw_direct(struct mmc_card *card, int write, unsigned fn,
            unsigned addr, u8 in, u8 * out);
    int mmc_io_rw_extended(struct mmc_card *card, int write, unsigned fn,
            unsigned addr, int incr_addr, u8 *buf, unsigned blocks, unsigned blksz);
    int sdio_reset(struct mmc_host *host);
```

这样整个命令操作以及传输功能就完整了。

7.3.3 TI 芯片 MMC 相关实现详解

DM 3730 的 MMC 控制器框架如图 7-11 所示。图 7-11 引自《DM 3730 芯片手册》中第 3398 页的框图。

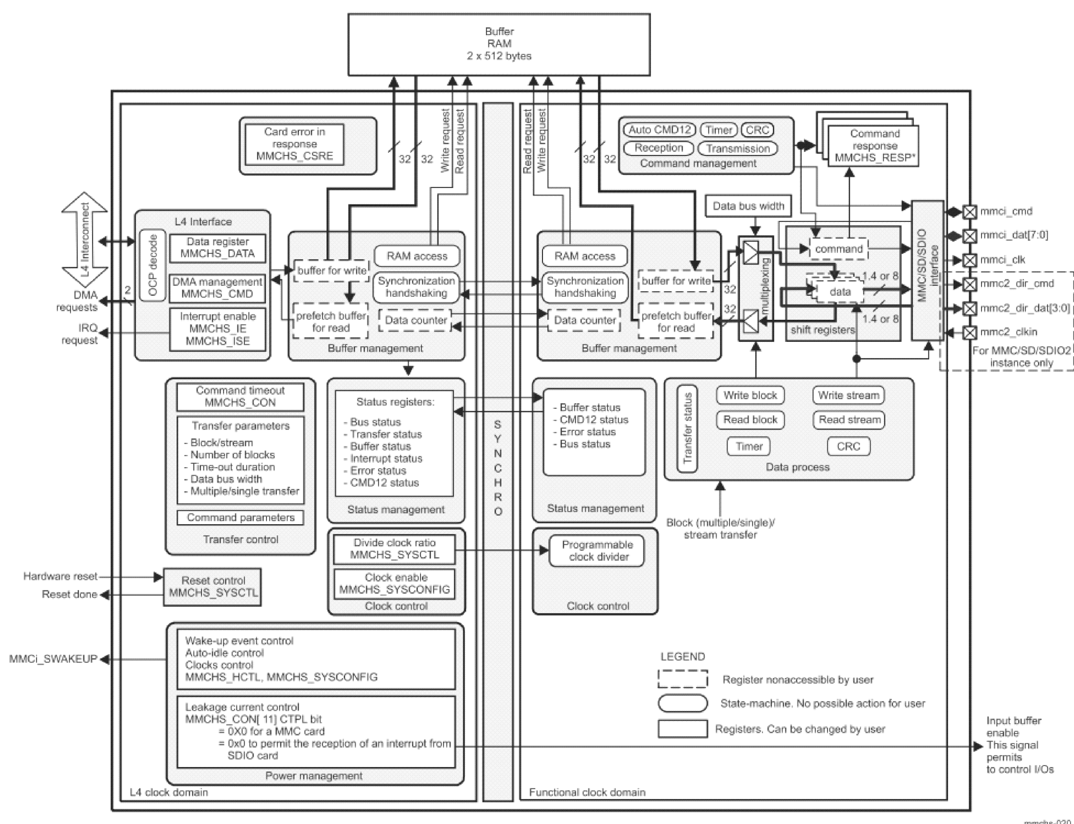


图 7-11 DM 3730 MMC 框架图

从图 7-11 可见，MMC 控制器中有复杂的命令和数据管理，控制器内部带有 buffer 用于数据缓冲，数据主要通过 DMA 读写。

关于 DM 3730 MMC 的驱动部分，主要分析其相关的初始化和总线传输以及 card 状态变化的操作。

1. 初始化

先来看看初始化部分，这里还是通过 platform driver 的 probe 函数来了解细节，其内容如下：

```
static int __init omap_hsmmc_probe(struct platform_device *pdev)
{
    struct omap_mmc_platform_data *pdata = pdev->dev.platform_data;
    struct mmc_host *mmc;
    struct omap_hsmmc_host *host = NULL;
    struct resource *res;
    int ret, irq;
```

```
if(pdata == NULL){
    dev_err(&pdev -> dev, "Platform Data is missing\n");
    return -ENXIO;
}
//如果没有槽一定是错的
if(pdata -> nr_slots == 0){
    dev_err(&pdev -> dev, "No Slots\n");
    return -ENXIO;
}
//获得寄存器地址信息
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
//获得中断号
irq = platform_get_irq(pdev, 0);
if(res == NULL || irq < 0)
    return -ENXIO;

//绑定相应的寄存器空间.
res -> start += pdata -> reg_offset;
res -> end += pdata -> reg_offset;
res = request_mem_region(res -> start, res -> end - res -> start + 1, pdev -> name);
if(res == NULL)
    return -EBUSY;

//card detect 的 GPIO 相关的初始化,如没有相应 GPIO 作为 card 检测
//则该操作实际为空
ret = omap_hsmmc_gpio_init(pdata);
if(ret)
    goto err;

//分配相应的 mmc 框架层的 mmc host,以及设备特殊结构 omap_hsmmc_host
//为附加信息
mmc = mmc_alloc_host(sizeof(struct omap_hsmmc_host), &pdev -> dev);
if(!mmc){
    ret = -ENOMEM;
    goto err_alloc;
}

//对于 mmc 层的 mmc host 结构附加的 omap_hsmmc_host 进行初始化
host        = mmc_priv(mmc);
//建立 mmc host 和 omap hsmmc 的关联
host -> mmc     = mmc;
//platform special data 和 slot 相关
```

```
	host -> pdata      = pdata;
	host -> dev        = &pdev -> dev;
	//使用 DMA
	host -> use_dma       = 1;
	//DMA 地址范围
	host -> dev -> dma_mask = &pdata -> dma_mask;
	host -> dma_ch = -1;
	//irq 号
	host -> irq = irq;
	//记录 MMC 接口 ID
	host -> id = pdev -> id;
	//每个 MMC controller 只接一个 slot
	host -> slot_id  = 0;
	//寄存器物理基地址
	host -> mapbase       = res -> start;
	//映射基地址
	host -> base       = ioremap(host -> mapbase, SZ_4K);
	host -> power_mode = MMC_POWER_OFF;

	//设置私有的 drv data 便于 pm 部分代码使用
	platform_set_drvdata(pdev, host);
	//创建 work,work 的执行实体为 omap_hsmmc_detect
	INIT_WORK(&host -> mmc_carddetect_work, omap_hsmmc_detect);

	//根据参数设置挂载不同的控制器操作接口
	if(mmc_slot(host). power_saving)
		mmc -> ops       = &omap_hsmmc_ps_ops;
	else
		mmc -> ops       = &omap_hsmmc_ops;

	if(mmc_slot(host). vcc_aux_disable_is_sleep)
		mmc_slot(host). no_off = 1;

	//总线频率的范围
	mmc -> f_min = 400000;
	mmc -> f_max = 52000000;

	spin_lock_init(&host -> irq_lock);

	//获得时钟
	host -> iclk = clk_get(&pdev -> dev, "ick");
…
```

```
        host -> fclk = clk_get(&pdev -> dev, "fck");
...
        //MMC 上下文保存
        omap_hsmmc_context_save(host);

        //设置 MMC 可以 disable 的能力
        mmc -> caps |= MMC_CAP_DISABLE;
        //设置 mmc 层 host controller 超时多少之后可以 disable controller
        mmc_set_disable_delay(mmc, OMAP_MMC_DISABLED_TIMEOUT);
        /* we start off in DISABLED state */
        //设置 host controller 的 power 状态为 disable
        host -> dpm_state = DISABLED;

        //enable 必需的 clock
        if(clk_enable(host -> iclk) != 0) {
            clk_put(host -> iclk);
            clk_put(host -> fclk);
            goto err1;
        }
        //使能控制器
        if(mmc_host_enable(host -> mmc) != 0) {
            clk_disable(host -> iclk);
            clk_put(host -> iclk);
            clk_put(host -> fclk);
            goto err1;
        }
...
        /* Since we do only SG emulation, we can have as many segs
         * as we want. */
        //下面的参数是 block layer 相关的参数,对于优化性能是有意义的
        mmc -> max_segs = 1;

        mmc -> max_blk_size = 512;       /* Block Length at max can be 1024 */
        mmc -> max_blk_count = 0xFFFF;    /* No. of Blocks is 16 bits */
        mmc -> max_req_size = mmc -> max_blk_size * mmc -> max_blk_count;
        mmc -> max_seg_size = mmc -> max_req_size;

        //mmc 相关的能力属性设置,这里主要是设置高速 mmc/sd high speed
mmc -> caps |= MMC_CAP_MMC_HIGHSPEED | MMC_CAP_SD_HIGHSPEED |
MMC_CAP_WAIT_WHILE_BUSY | MMC_CAP_ERASE;

        mmc -> caps |= mmc_slot(host).caps;
```

```
	if(mmc -> caps & MMC_CAP_8_BIT_DATA)
		mmc -> caps |= MMC_CAP_4_BIT_DATA;

	//如果插槽中的设备不能移动则设置相应属性
	if(mmc_slot(host).nonremovable)
		mmc -> caps |= MMC_CAP_NONREMOVABLE;

	mmc -> pm_caps |= MMC_PM_KEEP_POWER;

	omap_hsmmc_conf_bus_power(host);

	//获得 DMA 信息
	res = platform_get_resource(pdev, IORESOURCE_DMA, 0);
	if(!res)
		goto err1;
	host -> dma_line_rx = res -> start;
	res = platform_get_resource(pdev, IORESOURCE_DMA, 1);
	if(!res)
		goto err1;
	host -> dma_line_tx = res -> start;

	/* Request IRQ for MMC operations */
	//相关中断初始化
	ret = request_irq(host -> irq, omap_hsmmc_irq, IRQF_DISABLED,
			mmc_hostname(mmc), host);
	if(ret){
		dev_dbg(mmc_dev(host -> mmc), "Unable to grab HSMMC IRQ\n");
		goto err_irq;
	}
...
	/* Request IRQ for card detect */
	//card detect 中断信号
	if((mmc_slot(host).card_detect_irq)){
		ret = request_irq(mmc_slot(host).card_detect_irq, omap_hsmmc_cd_handler, IRQF_TRIG-
GER_RISING | IRQF_TRIGGER_FALLING | IRQF_DISABLED,  mmc_hostname(mmc), host);
		if(ret){
			dev_dbg(mmc_dev(host -> mmc),
				"Unable to grab MMC CD IRQ\n");
			goto err_irq_cd;
		}
		pdata -> suspend = omap_hsmmc_suspend_cdirq;
		pdata -> resume = omap_hsmmc_resume_cdirq;
```

```
        }

        omap_hsmmc_disable_irq(host);
        mmc_host_lazy_disable(host->mmc);
        //对 card 进行保护,避免不正确的写损坏卡
        omap_hsmmc_protect_card(host);

        //向 MMC 框架注册总线控制器,其后会进行设备发现相关操作
        mmc_add_host(mmc);

        //创建 sysfs 中操作接口
        if(mmc_slot(host).name! = NULL){
            ret = device_create_file(&mmc->class_dev, &dev_attr_slot_name);
            if(ret <0)
                goto err_slot_name;
        }
        if(mmc_slot(host).card_detect_irq && mmc_slot(host).get_cover_state){
            ret = device_create_file(&mmc->class_dev, &dev_attr_cover_switch);
            if(ret <0)
                    goto err_slot_name;
        }

        omap_hsmmc_debugfs(mmc);
        return 0;
    …
    }
```

从分析中可见，主要的工作是进行属性设置、资源申请和接口注册，然后通过 mmc_add_host 的调用实现功能。

相应的 platform device 是在系统初始化时通过 omap_mmc_add 进行的，这里就不详述了。

2. 总线传输

总线传输的控制器接口是由 mmc_host_ops 中的 request 来定义的，DM 3730 MMC 控制器相应的操作是 omap_hsmmc_request。下面详细分析其功能：

```
    static void omap_hsmmc_request(struct mmc_host *mmc, struct mmc_request *req)
    {
        struct omap_hsmmc_host *host = mmc_priv(mmc);
        int err;

        BUG_ON(host->req_in_progress);
        BUG_ON(host->dma_ch! = -1);
```

```
        if(host->protect_card){
            //在保护阶段的传输请求都直接报错
            if(host->reqs_blocked < 3){
                omap_hsmmc_reset_controller_fsm(host, SRD);
                omap_hsmmc_reset_controller_fsm(host, SRC);
                host->reqs_blocked += 1;
            }
            req->cmd->error = -EBADF;
            if(req->data)
                req->data->error = -EBADF;
            req->cmd->retries = 0;
            mmc_request_done(mmc, req);
            return;
        } else if(host->reqs_blocked)
            host->reqs_blocked = 0;
        WARN_ON(host->mrq != NULL);
        //请求由控制器驱动接管
        host->mrq = req;
        //这里做数据的准备工作,如果有数据操作,会对 DMA 进行初始化,
        //并启动 DMA channel,在需要进行 DMA 传输时控制器会发送 request 信号
        //开始实际的传输,这是硬件自动完成的
        err = omap_hsmmc_prepare_data(host, req);
        if(err){
            req->cmd->error = err;
            if(req->data)
                req->data->error = err;
            host->mrq = NULL;
            mmc_request_done(mmc, req);
            return;
        }
        //总线控制器启动总线传输
        omap_hsmmc_start_command(host, req->cmd, req->data);
    }
```

从中可见，主要是进行操作前的准备工作，如果有数据传输操作，则需要在函数 omap_hsmmc_prepare_data 中进行 DMA 申请、配置、启动等完成相关的准备工作，剩下的就是等待硬件信号来完成数据部分的操作。

总线控制器的传输启动操作是通过 omap_hsmmc_start_command 来实现的，详细分析如下：

```
    static void
    omap_hsmmc_start_command(struct omap_hsmmc_host *host, struct mmc_command *cmd,
```

```
        struct mmc_data  * data)
{
    int cmdreg = 0, resptype = 0, cmdtype = 0;

    dev_dbg(mmc_dev(host->mmc), "%s: CMD%d, argument 0x%08x\n",
        mmc_hostname(host->mmc), cmd->opcode, cmd->arg);
    //操作命令
    host->cmd = cmd;
    //使能中断
    omap_hsmmc_enable_irq(host, cmd);
    host->response_busy = 0;
    //根据命令类型中的标记进行总线设备响应类型等参数的设置
    if(cmd->flags & MMC_RSP_PRESENT){
        if(cmd->flags & MMC_RSP_136)
            resptype = 1;
        else if(cmd->flags & MMC_RSP_BUSY){
            resptype = 3;
            host->response_busy = 1;
        } else
            resptype = 2;
    }

    if(cmd == host->mrq->stop)
        cmdtype = 0x3;
    //进行寄存器值的基本设置,在操作中控制器会对响应类型等进行检查
    cmdreg = (cmd->opcode << 24) | (resptype << 16) | (cmdtype << 22);
    //如果有数据传输进行相应的设置
    if(data){
        cmdreg |= DP_SELECT | MSBS | BCE;
        if(data->flags & MMC_DATA_READ)
            cmdreg |= DDIR;
        else
            cmdreg &= ~(DDIR);
    }
    //如果使用 DMA 则进行属性设置,保证控制器可以发送 DMA 请求信号
    if(host->use_dma)
        cmdreg |= DMA_EN;
    //记录控制器在处理状态
    host->req_in_progress = 1;
    //写寄存器开始操作
    OMAP_HSMMC_WRITE(host->base, ARG, cmd->arg);
    OMAP_HSMMC_WRITE(host->base, CMD, cmdreg);
}
```

主要是根据传输的命令以及状态进行寄存器设置。接下来控制器会根据操作执行的情况上报中断，中断处理函数 omap_hsmmc_irq 会进行状态的检查，根据状态做必要处理。如进行数据操作时有错误，则需要进行必要的 DMA 清理，最终通过 mmc_request_done 来向 MMC 框架层报告，mmc request 的操作完整状态信息都在 mmc_request 中。这样就完成了整个的传输操作。

3. card 状态变化

MMC 总线设备有插拔的可能，这就需要能够检测设备的状态，下面来看看具体如何实现。在初始化时，已经见到一个检测相应信号的中断，当卡的状态发生变化时会有中断产生，在实现中由 omap_hsmmc_cd_handler 进行处理，具体如下：

```
//该函数在 kernel 中提供 card detect 的处理,对于该 handler 某些 chip 的
//mmc controller 通过单独 GPIO irq 处理,某些 chip 通过 mmc controller 的
//统一 irq 检测调用。具体和 chip 相关,在没有单独的 GPIO 作为 card detect 的
//chip 如 dm816x 通过 mmc 统一的 irq handler 中调用该 handler 完成后续处理
static irqreturn_t omap_hsmmc_cd_handler(int irq, void *dev_id)
{
    struct omap_hsmmc_host *host = (struct omap_hsmmc_host *)dev_id;

    if(host->suspended)
        return IRQ_HANDLED;
    //这里主要是 schedule card detect work,该 work 为 omap_hsmmc_detect
    //在 probe 函数中设置
    schedule_work(&host->mmc_carddetect_work);

    return IRQ_HANDLED;
}
```

可见通过 work 来进行延迟操作，具体分析如下：

```
//通过 work 的方式检查并通知 mmc core, card insert 事件该函数作为 work 的
//function 等待 schedule 调度,优点是延时调用 mmc_detect_change 以避免
//不完整 insert/remove 等操作
static void omap_hsmmc_detect(struct work_struct *work)
{
    struct omap_hsmmc_host *host =
        container_of(work, struct omap_hsmmc_host, mmc_carddetect_work);
    struct omap_mmc_slot_data *slot = &mmc_slot(host);
    int carddetect;

    if(host->suspended)
        return;
```

```
        //通知应用层 sysfs 的 poll 相应属性的应用,相应的属性
        //发生变化,这里属性变化主要是说明 card 的拔插. 对应
        //于 kernel 开发的 sysfs 属性的 poll 功能,留给应用侦测接口
        sysfs_notify(&host->mmc->class_dev.kobj, NULL, "cover_switch");

        //如果 slot 接口有 card_detect,则说明有 GPIO 执行通知操作
        //或者有寄存器接口查询 card detect 情况
        if(slot->card_detect)
            carddetect = slot->card_detect(host->dev, host->slot_id);
        else {
            //如果没有 card detect 操作为了保护 card,需要进行 card 的
            //保护操作,避免破坏 card
            omap_hsmmc_protect_card(host);
            carddetect = -ENOSYS;
        }

        //通知 mmc core 层相应的 host 检测到变化,延时后由 core 层
        //进一步检查并进行后续操作
        if(carddetect)
            mmc_detect_change(host->mmc,(HZ * 200) / 1000);
        else
            mmc_detect_change(host->mmc,(HZ * 50) / 1000);
    }
```

主要的操作就是获得当前卡的状态，并执行 mmc_detect_change，但是会根据状态的不同进行延时后再执行设备发现操作。

具体的卡状态检测由 omap_hsmmc_card_detect 来执行，具体分析如下：

```
    //card detect 的平台接口函数,主要是作为平台的 card detect 接口函数
    //注册为相应 host slot 接口的 card_detect 接口进行调用,由 schedule work
    //omap_hsmmc_detect 函数调用. 该函数返回 card detect 的状态,通过这里
    //的延时检查保证不会有 jitter 的问题
    static int omap_hsmmc_card_detect(struct device *dev, int slot)
    {
        struct omap_mmc_platform_data *mmc = dev->platform_data;
        struct omap_hsmmc_host *host =
            platform_get_drvdata(to_platform_device(dev));

        //返回 card detect 的状态
        if(mmc->version != MMC_CTRL_VERSION_2)
            /* NOTE: assumes card detect signal is active-low */
            return !gpio_get_value_cansleep(mmc->slots[0].switch_pin);
```

```
        else {
            u32 pstate = 0;
            u32 enabled = 0;
            //这里是 dm81xx 芯片使用的 mmc controller ip

            enabled = host -> mmc -> enabled;
            //如果没有 enable,要先 enable 才能读取寄存器
            if( ! enabled)
                mmc_host_enable( host -> mmc) ;

            pstate = OMAP_HSMMC_READ( host -> base, PSTATE) ;

            //保证读取状态寄存器后恢复 controller 的状态
            if( ! enabled)
                mmc_host_disable( host -> mmc) ;
            printk( "PSTATE %x\n" , pstate) ;
            pstate = pstate & PSTATE_CINS_MASK;
            pstate = pstate >> PSTATE_CINS_SHIFT;
            return pstate;
        }
    }
```

这样整个状态变化的操作就完整了。

7.3.4 MMC 电源管理相关说明

对于 MMC 整体的电源管理，首先来看总线部分，在 mmc_bus_type 中与电源管理相关的操作接口如下：

```
    static struct bus_type mmc_bus_type = {
    …
        . suspend = mmc_bus_suspend,
        . resume          = mmc_bus_resume,
        . pm         = MMC_PM_OPS_PTR,
    } ;
```

可见其中实现了 SLM 以及 runtime pm 的接口。SLM 以 suspend 操作为例，细节如下：

```
    static int mmc_bus_suspend( struct device  * dev, pm_message_t state)
    {
        struct mmc_driver  * drv = to_mmc_driver( dev -> driver) ;
        struct mmc_card  * card = mmc_dev_to_card( dev) ;
```

```
        int ret = 0;

        if(dev -> driver && drv -> suspend)
            ret = drv -> suspend(card, state);
        return ret;
    }
```

可见主要是执行驱动的 suspend 操作，在 MMC 框架中就是 mmcblk 驱动，相应的接口是 mmc_blk_suspend，其中实际的操作就是通过 blk_stop_queue 来停止接收块操作请求。

以 runtime suspend 的接口为例讲解 runtime pm 的接口。在 MMC 框架中接口是 mmc_runtime_suspend，其会通过 mmc_power_save_host 来调用 mmc_bus_ops 中的 power_save 接口，进行 runtime pm 的操作。实际中 power_save 接口并没有进行设置。

另外在实际操作过程中为了减少功耗，MMC 框架提供了 mmc_host_enable 和 mmc_host_disable 接口，用于总线控制器的电源管理操作，在需要时 enable，不需要时 disable。这样可以降低控制器的功耗，而 mmc_host_ops 中需要提供 enable 和 disable 接口进行相应的操作。

总线控制器的电源管理操作，主要是由 platform driver 提供的。DM 3730 的 MMC 总线控制器相应的操作如下：

```
    static struct dev_pm_ops omap_hsmmc_dev_pm_ops = {
        .suspend = omap_hsmmc_suspend,
        .resume         = omap_hsmmc_resume,
    };
```

以 suspend 操作为例进行分析，细节如下：

```
    static int omap_hsmmc_suspend(struct device *dev)
    {
        int ret = 0;
        struct platform_device *pdev = to_platform_device(dev);
        struct omap_hsmmc_host *host = platform_get_drvdata(pdev);

        if(host && host -> suspended)
            return 0;
        if(host) {
            //进入 suspend 状态
            host -> suspended = 1;
            //这里如果 card detect 是 GPIO 实现,则需要清理,并做必要的处理
            if(host -> pdata -> suspend) {
                ret = host -> pdata -> suspend(&pdev -> dev, host -> slot_id);
                …
            }
```

```
        //取消 card detect 相关操作
        cancel_work_sync(&host->mmc_carddetect_work);
        //进行 MMC 层面的总线控制器 suspend 操作,这里是执行 bus ops
        //中的 suspend 操作,不同的规范操作不同,主要是设置状态,对于 SDIO
        //则会执行驱动的 suspend 操作
        ret = mmc_suspend_host(host->mmc);
        mmc_host_enable(host->mmc);
        if(ret == 0) {
            omap_hsmmc_disable_irq(host);
            //切断 bus power
            OMAP_HSMMC_WRITE(host->base, HCTL,
                    OMAP_HSMMC_READ(host->base, HCTL) & ~SDBP);
            mmc_host_disable(host->mmc);
            clk_disable(host->iclk);
            if(host->got_dbclk)
                clk_disable(host->dbclk);
        } else {
            //之前操作失败则恢复
            host->suspended = 0;
            if(host->pdata->resume) {
                ret = host->pdata->resume(&pdev->dev,
                                        host->slot_id);
                if(ret)
                    dev_dbg(mmc_dev(host->mmc),
                            "Unmask interrupt failed\n");
            }
            mmc_host_disable(host->mmc);
        }
    }
    return ret;
}
```

可见不仅对总线进行了具体操作 1 还将总线的 power 切断，做得还是很彻底的。

这样 MMC 总线的电源管理部分就基本完整了。

7.4　通用串行总线（USB）

7.4.1　USB 总线驱动需求

USB（Universal Serial Bus）是连接计算机系统与外部设备的一种串口总线标准，也是一种输入输出接口的技术规范，被广泛地应用于个人电脑和移动设备等通信产品，并扩展至摄影器材、数字电视（机顶盒）、游戏机等其他相关领域。

USB 最初是由英特尔公司与微软公司倡导发起的，其最大的特点是支持热插拔和即插即用。当设备插入时，主机枚举到此设备并加载所需的驱动程序，使用比其他总线方便。在速度方面，USB 1.1 的最大传输带宽为 12 Mbit/s，USB 2.0 的最大传输带宽为 480 Mbit/s，USB 3.0 的最大传输带宽为 5 Gbit/s。

USB 的设计为非对称式的，它是主从式总线，任何 USB 事务都是由主机引发的。USB 主机处于主模式，设备处于从模式。USB 总线是由主机控制器和若干 hub 设备以及与 hub 连接的从设备组成的。USB 总线拓扑如图 7-12 所示。图 7-12 引自《USB2.0 规范》。

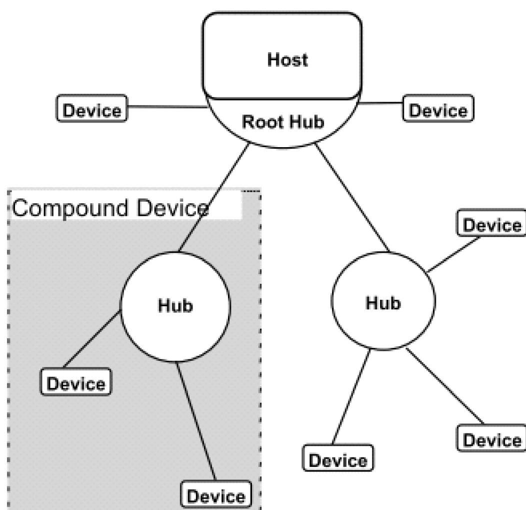


图 7-12　USB 总线拓扑

从图 7-12 可见，一个 Host 控制器会和 root hub 绑定，并可以通过 Hub 连接多个设备。

USB 可以连接的外设有鼠标、键盘、游戏手柄、游戏杆、扫描仪、数码相机、打印机、硬盘和网络部件等各种设备。USB 总线已经成为使用最广泛的设备连接标准。

规范中为了实现以上的功能对总线中的很多环节都进行了定义。一个 USB 物理设备可以承担多种功能，USB 的术语中设备（device）指的是功能（functions），更注重逻辑性。规范中关于主从设备的交互如图 7-13 所示。

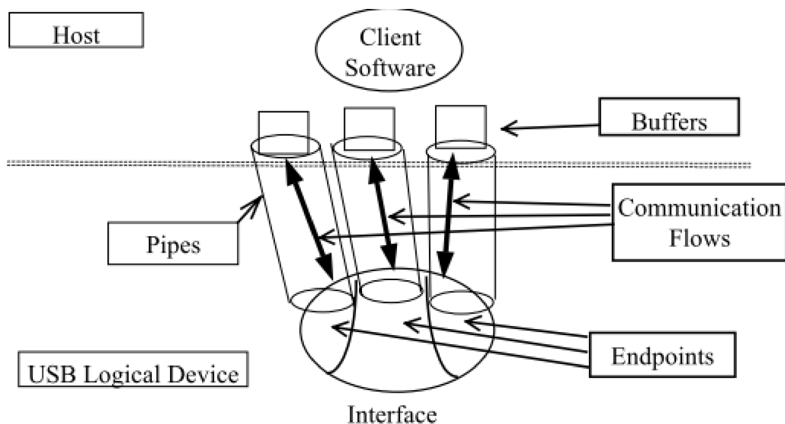


图 7-13　USB 主从设备交互框图

从图 7-13 可见，管道（pipe）把主机控制器和端点（endpoint）连接起来形成一个交互通道进行主从设备的交互。端点只能单向（进/出）传输数据，管道也是单向的。每个 USB 设备至少有两个端点/管道，分别是进和出两个方向，编号为 0，用于控制总线上的设备。按照各自的传输类型，管道被分为 4 类：

- 控制传输（Control）。一般用于短的、简单的设备命令和状态反馈，例如用于总线控制的 0 号管道。
- 同步传输（Isochronous）。按照有保障的速度传输，可能有数据丢失，例如实时的音频和视频。
- 中断传输（Interrupt）。用于必须保证尽快反应的设备（有限延迟），例如鼠标和键盘。
- 批量传输（Bulk）。使用余下的带宽大量地（但是没有对于延迟、连续性、带宽和速度的保证）传输数据，例如普通的文件传输。

为了访问端点，必须获得设备一个分层的配置。连接到主机的设备只有一个设备描述符（device descriptors），而设备描述符有若干配置描述符（configuration descriptors）。这些配置一般与状态相对应，例如活跃和节能模式。每个配置描述符有若干接口描述符（interface setting），用于描述设备的一个功能方面，所以可以用于描述从设备不同的功能，如一个相机可能拥有视频和音频两个接口。接口描述符有一个默认接口设置（default interface settings）和可能多个替代接口设置（alternate interface settings），接口设置中包含端点描述符。一个端点能够在多个接口和替代接口设置之间复用。

总体上来说，系统对 USB 总线驱动的需求就是要能按规范实现总线的各种功能，并且满足总线的各种无关性的需求。

7.4.2 USB 总线驱动框架解析

1. 总线相关以及核心框架

针对 USB 功能，Linux 内核中提供了主设备和从设备两个不同的框架，这里主要介绍作为主设备的框架。USB 总线主设备框架还是以 bus_type 作为入口来了解总线框架。

```
struct bus_type usb_bus_type = {
    .name =        "usb",
    .match =    usb_device_match,
    .uevent =    usb_uevent,
#ifdef CONFIG_USB_SUSPEND
    .pm =        &usb_bus_pm_ops,
#endif
};
```

usb_bus_type 中重要的是 usb_device_match，下面来了解细节：

```
static int usb_device_match(struct device *dev, struct device_driver *drv)
{
    /* devices and interfaces are handled separately */
```

```
        //对 device 进行区分
        if(is_usb_device(dev)){
            //这里表明是 usb device
            /* interface drivers never match devices */
            //检查驱动是否对应于 usb device
            if(!is_usb_device_driver(drv))
                return 0;
            return 1;
        } else if(is_usb_interface(dev)){
            //这里是 usb interface
            struct usb_interface *intf;
            struct usb_driver *usb_drv;
            const struct usb_device_id *id;

            /* device drivers never match interfaces */
            if(is_usb_device_driver(drv))
                return 0;
            //这里驱动和设备类型是匹配的
            intf = to_usb_interface(dev);
            usb_drv = to_usb_driver(drv);
            //进行真正的匹配,匹配 ID 表
            id = usb_match_id(intf, usb_drv->id_table);
            if(id)
                return 1;
            //匹配动态 ID
            id = usb_match_dynamic_id(intf, usb_drv);
            if(id)
                return 1;
        }
        return 0;
    }
```

从分析中可知，USB 总线框架主要管理的是两类设备，分别是 usb_device 和 usb_interface。从 USB 规范的角度可知 usb_interface 更专注于功能，应该是获得描述符之后才创建的，而 usb_device 则对应于 USB 物理设备。这样就从逻辑功能和物理层面都对设备进行管理，从管理层次上来说也是完整的。对应的驱动包括 usb_device_driver 和 usb_driver，其中 usb_device_driver 对应于 usb_device，而 usb_driver 则对应于 usb_interface。usb_device_driver 主要用于实现设备发现，而 usb_driver 则主要实现与功能模块的关联以及电源管理相关的功能。

像其他总线设备一样，总线控制器负责总线事务的交互。USB 主设备总线框架就是对以上内容的管理，具体架构如图 7-14 所示。

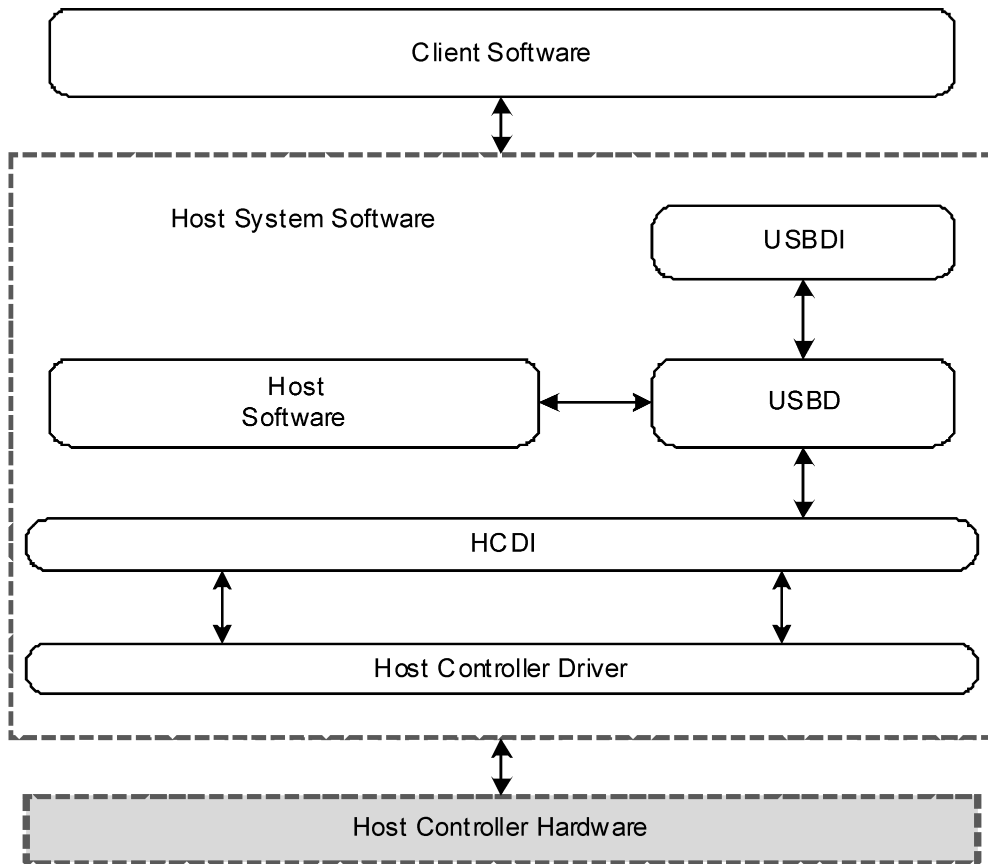


图 7-14　Linux USB 主设备总线架构

从图 7-14 可见，USB 总线框架各模块划分，包括 host software 主要负责协议相关的部分，包括枚举设备、数据传输等；USBDI（USB driver interface）主要负责屏蔽 USB 细节，为其他功能框架使用 USB 总线传输的接口层；USBD（即 USB driver）负责 USB 设备的功能关联；HCDI 则是屏蔽不同总线控制器之间的差别；最后 Host Controller Driver 是具体的总线控制器的驱动，相应地负责与总线控制器交互完成总线传输。

这样整体框架从层次的角度就完整了。

2. 总线控制器相关

USB 总线框架对总线控制器的管理主要是通过 usb_hcd 来实现的，其详细分析如下：

```
struct usb_hcd {
    //总线控制器所管理的总线信息,其中包含总线编号以及 root hub 的信息
    struct usb_bus          self;           /* hcd is - a bus */
    struct kref             kref;           /* reference counter */

    const char              * product_desc; /* product/vendor string */
    char                    irq_descr[24];  /* driver + bus # */
    //root hub 需要的 timer
```

```
        struct timer_list       rh_timer;       /* drives root-hub polling */
        //通常为 root hub 的状态 urb 用于设备发现
        struct urb              *status_urb;    /* the current status urb */
#ifdef CONFIG_USB_SUSPEND
        struct work_struct      wakeup_work;    /* for remote wakeup */
#endif
        //omap3 的 work 操作用于电源管理
        struct work_struct ehci_omap_work;
        //总线传输的操作接口
        const struct hc_driver  *driver;        /* hw-specific hooks */
        //总线控制器的各种属性
        unsigned long           flags;

        /* Flags that get set only during HCD registration or removal. */
        //各种特性信息
        unsigned                rh_registered:1;/* is root hub registered? */
        unsigned                rh_pollable:1;  /* may we poll the root hub? */

        unsigned                uses_new_polling:1;
        unsigned                wireless:1;     /* Wireless USB HCD */
        unsigned                authorized_default:1;
        unsigned                has_tt:1;       /* Integrated TT in root hub */
        //分配的中断
        int                     irq;            /* irq allocated */
        //寄存器映射的起始地址
        void __iomem            *regs;          /* device memory/io */
        u64                     rsrc_start;     /* memory/io resource start */
        u64                     rsrc_len;       /* memory/io resource length */
        unsigned                power_budget;   /* in mA, 0 = no limit */
        struct mutex            bandwidth_mutex;
        //缓冲空间
        struct dma_pool         *pool[HCD_BUFFER_POOLS];
        //状态
        int                     state;
        //为不同类型的总线控制器分配的空间
        unsigned long hcd_priv[0] __attribute__((aligned(sizeof(unsigned long))));
};
```

从管理实体中可见，其中除了自身的属性和状态之外还包含了 root hub 相关的信息，这与硬件及规范一致。另外重要的就是 hc_driver，其中包含实际操作的接口，负责总线传输。由于 USB 总线控制器有不同的类型，如 OHCI、EHCI、UHCI、XHCI 等，这里 usb_hcd 是对所有这些总线控制器共性的抽象，相应的 hcd_priv 指向不同控制器私有的属性，主要是定义

不同的传输排列方式以及控制状态寄存器等。这里与其他总线控制器实体不同的是没有设备模型的实体，主要是由于 USB 总线控制器与 root hub 的亲密关系，而 root hub 本身就是设备，而设备的层次关系通过 usb_bus 中的 controller 进行关联。

关于总线控制器，USB 框架提供了创建以及注册管理的接口。首先来了解创建的接口 usb_create_hcd：

```
struct usb_hcd *usb_create_hcd(const struct hc_driver *driver,
        struct device *dev, const char *bus_name)
{
    struct usb_hcd *hcd;
    //分配控制器的整体空间,包括公共的 hcd 空间和具体总线控制器的私有空间,
    //私有空间的大小由 hc_driver 中的 hcd_priv_size 来定义
    hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, GFP_KERNEL);
    if(!hcd){
        dev_dbg(dev, "hcd alloc failed\n");
        return NULL;
    }
    dev_set_drvdata(dev, hcd);
    kref_init(&hcd->kref);
    //初始 USB 总线以及相关信息
    usb_bus_init(&hcd->self);
    hcd->self.controller = dev;
    hcd->self.bus_name = bus_name;
    hcd->self.uses_dma = (dev->dma_mask != NULL);
    //初始 root hub 定时操作等信息
    init_timer(&hcd->rh_timer);
    hcd->rh_timer.function = rh_timer_func;
    hcd->rh_timer.data = (unsigned long) hcd;
#ifdef CONFIG_USB_SUSPEND
    INIT_WORK(&hcd->wakeup_work, hcd_resume_work);
#endif
    mutex_init(&hcd->bandwidth_mutex);
    //设置总线操作接口
    hcd->driver = driver;
    hcd->product_desc = (driver->product_desc) ? driver->product_desc :
            "USB Host Controller";

    /* ehci omap specific */
    if(hcd->driver->recover_hcd)
        INIT_WORK(&hcd->ehci_omap_work, hcd->driver->recover_hcd);
    return hcd;
}
```

从分析中可见，USB 总线控制器的操作已经与 USB 总线以及 root hub 的操作相结合，这与它们之间的紧密关系是分不开的。

对于注册与管理接口，USB 框架的接口为 usb_add_hcd，主要功能是初始化并注册总线控制器，其中同样包括注册 USB 总线以及创建并注册 root hub。在调用了 usb_add_hcd 时，就创建了总线，从而可以进行设备发现以及驱动功能绑定的工作。

框架提供了统一的传输以及其他总线控制器功能操作接口，具体如下：

```
int usb_hcd_submit_urb(struct urb *urb, gfp_t mem_flags);
int usb_hcd_unlink_urb(struct urb *urb, int status);
void usb_hcd_giveback_urb(struct usb_hcd *hcd, struct urb *urb, int status);
void unmap_urb_setup_for_dma(struct usb_hcd *, struct urb *);
void unmap_urb_for_dma(struct usb_hcd *, struct urb *);
void usb_hcd_flush_endpoint(struct usb_device *udev, struct usb_host_endpoint *ep);
void usb_hcd_disable_endpoint(struct usb_device *udev, struct usb_host_endpoint *ep);
void usb_hcd_reset_endpoint(struct usb_device *udev, struct usb_host_endpoint *ep);
void usb_hcd_synchronize_unlinks(struct usb_device *udev);
int usb_hcd_alloc_bandwidth(struct usb_device *udev, struct usb_host_config *new_config,
        struct usb_host_interface *old_alt, struct usb_host_interface *new_alt);
int usb_hcd_get_frame_number(struct usb_device *udev);
```

这样总线控制器的底层与上层就完整了。

3. 总线设备相关

USB 设备在总线框架中已经有所介绍，主要由 usb_device 和 usb_interface 共同完成。先来了解 usb_device 的详细信息：

```
struct usb_device {
    //所在总线分配的地址
    int             devnum;
    //完整的路径
    char            devpath[16];
    //表示如何找到设备,其中包含通过 hub 以及 port 到达设备
    u32             route;
    //设备状态,主要用于枚举
    enum usb_device_state   state;
    //速度类型
    enum usb_device_speed   speed;
    //hub 用于不同类型控制器之间转换
    struct usb_tt   *tt;
    int             ttport;

    unsigned int toggle[2];
    //指向设备所在 hub,如果是 root hub 这里为空
```

```
    struct usb_device  * parent;
    //所在 bus,其中包含 root hub 信息
    struct usb_bus  * bus;
    //控制器 endpoint
    struct usb_host_endpoint ep0;

    //设备模型相关
    struct device dev;
    //描述符
    struct usb_device_descriptor descriptor;
    //配置
    struct usb_host_config  * config;
    //当前设置的配置
    struct usb_host_config  * actconfig;
    //所有的 endpoint 信息
    struct usb_host_endpoint  * ep_in[16];
    struct usb_host_endpoint  * ep_out[16];
    //原始的描述符信息
    char  * * rawdescriptors;

    //需要的电流
    unsigned short bus_mA;
    //所在 hub 的端口
    u8 portnum;
    //所在物理连接的层数
    u8 level;
    //状态信息
    unsigned can_submit:1;
    unsigned persist_enabled:1;
    unsigned have_langid:1;
    unsigned authorized:1;
    unsigned authenticated:1;
    unsigned wusb:1;
    int string_langid;
    /*  static strings from the device  */
    //设备信息
    char  * product;
    char  * manufacturer;
    char  * serial;
…
};
```

从分析中可见，总线规范中关于设备的各种属性都在 usb_device 中有了体现。再来看看 interface 的管理实体：

```
struct usb_interface {
    //可选的 interface 属性
    struct usb_host_interface *altsetting;
    //当前的 interface 属性
    struct usb_host_interface *cur_altsetting;    /* the currently active alternate setting */
    unsigned num_altsetting;    /* number of alternate settings */
    struct usb_interface_assoc_descriptor *intf_assoc;
    //如果 interface 直接针对功能设备,这里表示 usb class 的子设备号
    int minor;            /* minor number this interface is bound to */
    enum usb_interface_condition condition;        /* state of binding */
    //属性信息
    unsigned sysfs_files_created:1;    /* the sysfs attributes exist */
    unsigned ep_devs_created:1;    /* endpoint "devices" exist */
    unsigned unregistering:1;    /* unregistration is in progress */
    unsigned needs_remote_wakeup:1;    /* driver requires remote wakeup */
    unsigned needs_altsetting0:1;    /* switch to altsetting 0 is pending */
    unsigned needs_binding:1;    /* needs delayed unbind/rebind */
    unsigned reset_running:1;
    unsigned resetting_device:1;    /* true: bandwidth alloc after reset */
    //设备模型相关
    struct device dev;        /* interface specific device info */
    //针对 interface 直接对应用开放的功能,相应的为 class 类型设备
    struct device *usb_dev;
    atomic_t pm_usage_cnt;        /* usage counter for autosuspend */
    struct work_struct reset_ws;    /* for resets in atomic context */
};
```

usb_interface 会在枚举时创建 usb_host_interface，其中包含了 endpoint 的信息，这与总线规范是一致的，另外通过 interface_to_usbdev 可以关联到 usb_device（通过 dev 的 parent 关联的），这样设备层面就都关联在一起了。

总线设备最重要的功能就是设备发现，USB 总线的设备发现流程如图 7-15 所示。

这里主要是 hub 的功能，最终会通过 usb_new_device 来将 usb_device 注册进系统，相应的 usb_device 是通过 usb_alloc_dev 进行分配的，在 usb_new_device 中会读取描述符，在注册之后就会进行对应设备驱动的 match 工作。在总线框架中可知，usb_device 直接返回的值为 1，这是由于系统中只有一个对应于 usb_device 的驱动，即 usb_generic_driver，细节如下：

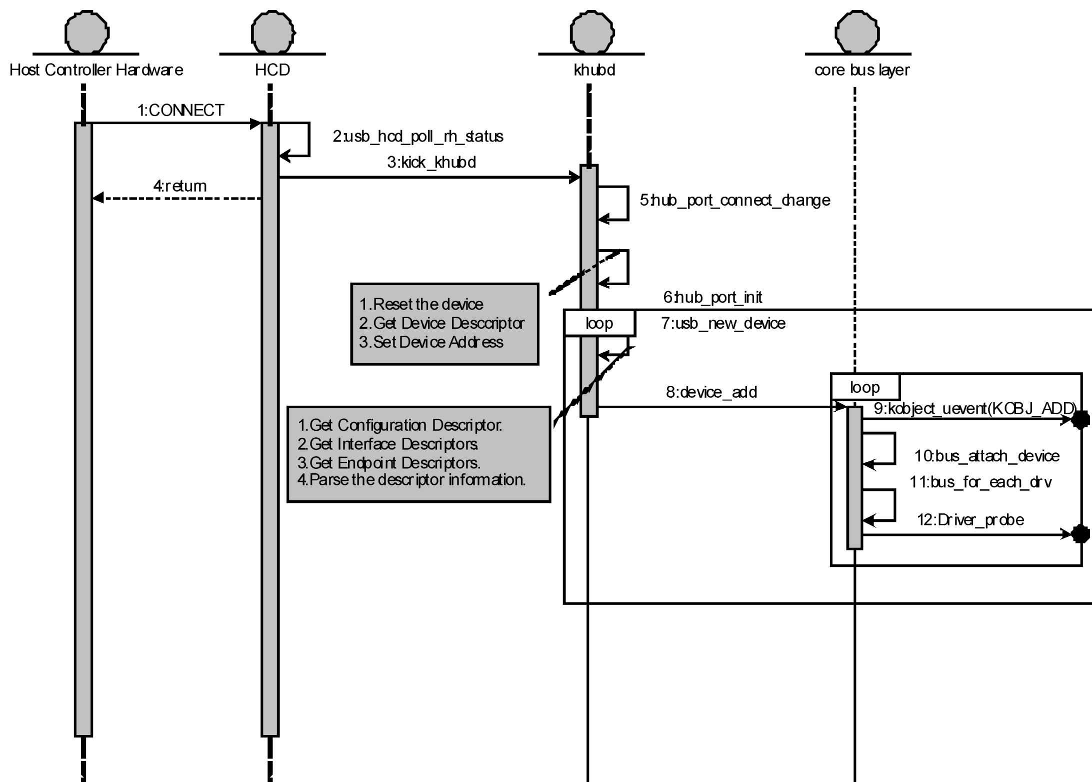


图 7-15　Linux USB 设备发现流程

```
struct usb_device_driver usb_generic_driver = {
        .name =        "usb",
        .probe = generic_probe,
        .disconnect = generic_disconnect,
#ifdef      CONFIG_PM
        .suspend = generic_suspend,
        .resume = generic_resume,
#endif
        .supports_autosuspend = 1,
};
```

接下来设备发现的功能由 generic_probe 实现，下面来看看细节：

```
static int generic_probe(struct usb_device *udev)
{
        int err, c;

        if(usb_device_is_owned(udev))
                ;           /* Don't configure if the device is owned */
```

```
        else if(udev -> authorized == 0)
            dev_err(&udev -> dev, "Device is not authorized for usage\n");
        else {
            //这里主要是进行选择配置,并进行配置的操作
            c = usb_choose_configuration(udev);
            if(c >= 0) {
                err = usb_set_configuration(udev, c);
                if(err) {
                    dev_err(&udev -> dev, "can't set config #%d, error %d\n", c, err);
                }
            }
        }
        /* USB device state == configured ... usable */
        //已经配置通知其他系统
        usb_notify_add_device(udev);
        return 0;
    }
```

这里主要是读取配置信息，并进行配置。在 usb_set_configuration 中会根据设备的属性创建所有的 usb_interface 并注册。这样 USB 系统就开始对功能驱动的匹配工作，相应的进入到 USB 总线 match 工作的后半部分，即 usb_interface 相关的部分，找到相应的驱动再执行 probe，从而完成整个设备发现及驱动绑定的功能。

4. 总线传输接口

USB 总线传输主要是实现 USB 总线规范中的四种传输，在 Linux USB 框架中通过 urb 进行管理，主要内容如下：

```
    struct urb {
        /* private: usb core and host controller only fields in the urb */
        struct kref kref;                   /* reference count of the URB */
        //特定总线控制器的信息
        void *hcpriv;                       /* private data for host controller */
        //并发提交次数
        atomic_t use_count;                 /* concurrent submissions counter */
        //失败次数
        atomic_t reject;                    /* submissions will fail */
        int unlinked;                       /* unlink error code */

        /* public: documented fields in the urb that can be used by drivers */
        //urb 链表
        struct list_head urb_list;          /* list head for use by the urb's current owner */
        struct list_head anchor_list;       /* the URB may be anchored */
        struct list_head giveback_list;     /* to postpone the giveback call */
```

```
    struct usb_anchor *anchor;
    //指向 USB 设备
    struct usb_device *dev;                 /*(in) pointer to associated device */
    //关联的 endpoint
    struct usb_host_endpoint *ep;           /*(internal) pointer to endpoint */
    //针对总线规范的 pipe 信息
    unsigned int pipe;                      /*(in) pipe information */
    //针对 Bulk
    unsigned int stream_id;                 /*(in) stream ID */
    int status;                             /*(return) non-ISO status */
    //表明 urb 如何被提交等传输属性
    unsigned int transfer_flags;            /*(in) URB_SHORT_NOT_OK |... */
    //相关的输入输出 buffer,与 endpoint 输入输出属性相关
    void *transfer_buffer;                  /*(in) associated data buffer */
    //DMA 操作时的空间,驱动可在 transfer_dma 与 transfer_buffer 之间选择
    dma_addr_t transfer_dma;                /*(in) dma addr for transfer_buffer */
    //buffer 的 scatterlist 表
    struct scatterlist *sg;                 /*(in) scatter gather buffer list */
    int num_sgs;                            /*(in) number of entries in the sg list */
    //buffer 的长度
    u32 transfer_buffer_length;             /*(in) data buffer length */
    //实际传输的长度
    u32 actual_length;                      /*(return) actual transfer length */
    unsigned char *setup_packet;            /*(in) setup packet(control only) */
    dma_addr_t setup_dma;                   /*(in) dma addr for setup_packet */
    int start_frame;                        /*(modify) start frame(ISO) */
    int number_of_packets;                  /*(in) number of ISO packets */
    int interval;                           /*(modify) transfer interval(INT/ISO) */
    int error_count;                        /*(return) number of ISO errors */
    //用于同步的参数
    void *context;                          /*(in) context for completion */
    //同步的回调
    usb_complete_t complete;                /*(in) completion routine */
    struct usb_iso_packet_descriptor iso_frame_desc[0];    /*(in) ISO ONLY */
};
```

USB 总线的传输是以 urb 为核心展开的。具体的流程如图 7-16 所示。

为了简化 USB 总线传输操作，USB 总线框架提供了如下接口以实现几种类型的总线传输功能：

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
        __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout);
```

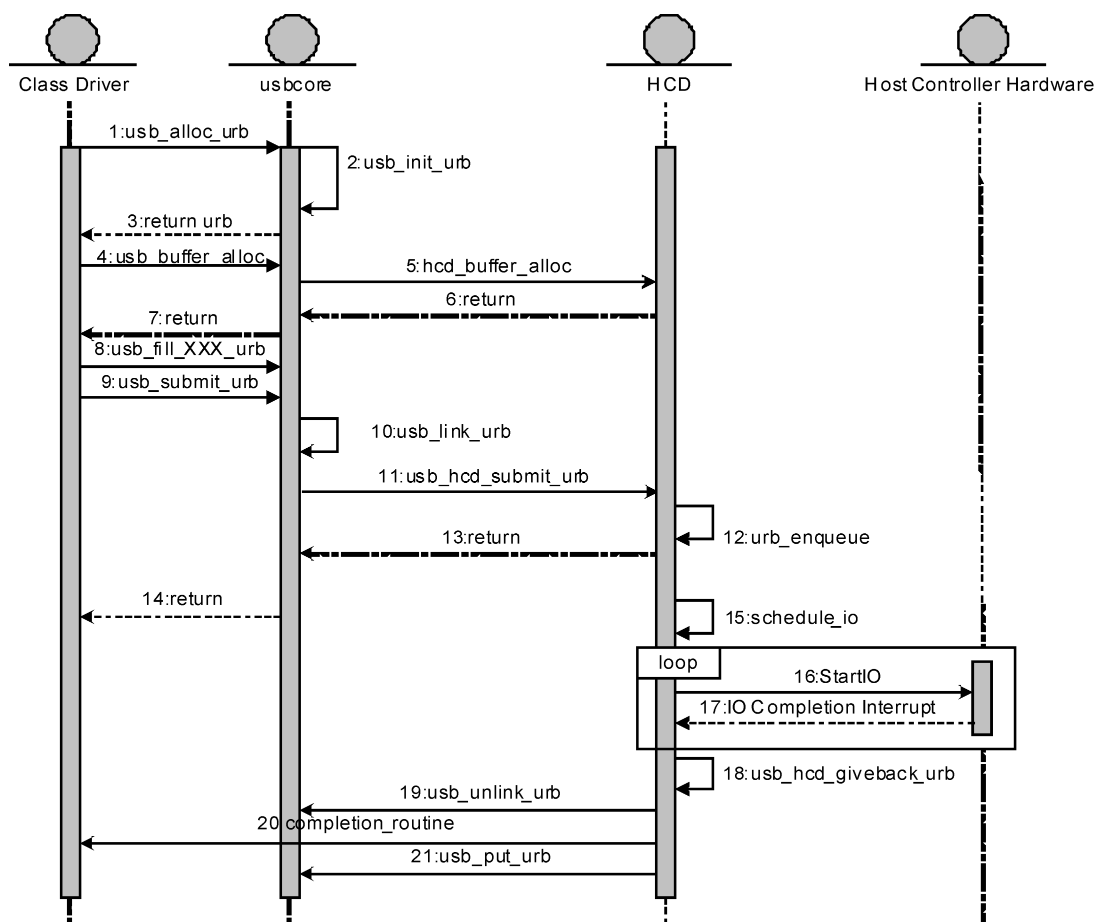


图 7-16　Linux USB 总线传输流程图

```
int usb_interrupt_msg(struct usb_device *usb_dev, unsigned int pipe,    void *data, int len,   int
 *actual_length, int timeout);
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,void *data, int len,
        int *actual_length, int timeout);
```

USB 总线框架已经为主要功能都提供了统一的操作，并提供了良好的接口供驱动开发。

7.4.3　TI 芯片 USB 总线驱动相关实现详解

DM 3730 的 USB 主机控制器框架如图 7-17 所示。图 7-17 引自《DM 3730 芯片手册》中第 3247 页的框图。

从图 7-17 中可见，DM 3730 中包含 OHCI 和 EHCI，另外是一些连接属性的配置，这里以 EHCI 为例进行介绍。

1. 初始化

先来看看初始化部分，还是通过 platform driver 的 probe 函数来了解细节，其内容如下：

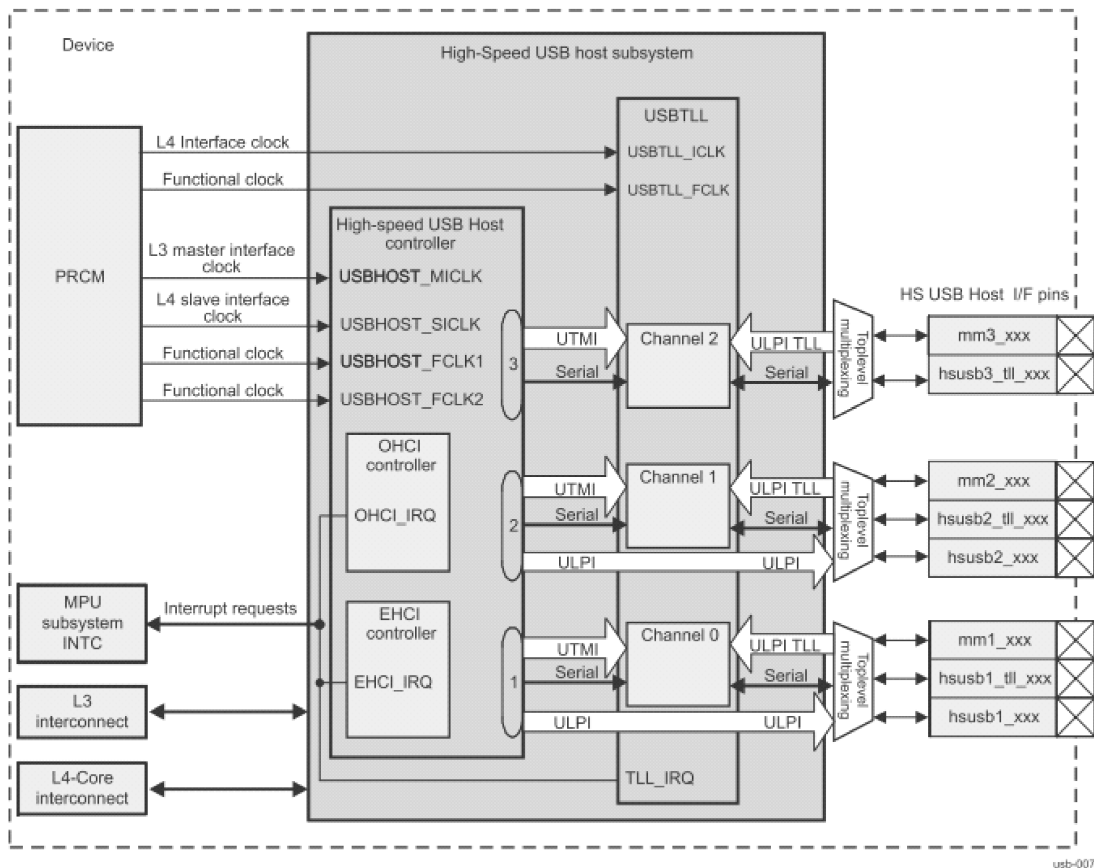


图 7-17　DM 3730 USB 主机控制器框架图

```
static int ehci_hcd_omap_probe(struct platform_device *pdev)
{
    struct ehci_hcd_omap_platform_data *pdata = pdev->dev.platform_data;
    struct ehci_hcd_omap *omap;
    struct resource *res;
    struct usb_hcd *hcd;
    //获得中断号
    int irq = platform_get_irq(pdev, 0);
    int ret = -ENODEV;
    int i;
    char supply[7];
…
    if(usb_disabled())
        goto err_disabled;
    //分配设备相关的管理实体
    omap = kzalloc(sizeof(*omap), GFP_KERNEL);
    if(!omap){
```

```
            ret = -ENOMEM;
            goto err_disabled;
        }
...
        //创建总线控制器,其中包含 EHCI 管理实体的空间
        ghcd = hcd = usb_create_hcd(&ehci_omap_hc_driver, &pdev->dev,
                    dev_name(&pdev->dev));
...
        //设置设备相关信息
        platform_set_drvdata(pdev, omap);
        omap->dev                   = &pdev->dev;
        omap->phy_reset             = pdata->phy_reset;
        omap->reset_gpio_port[0]    = pdata->reset_gpio_port[0];
        omap->reset_gpio_port[1]    = pdata->reset_gpio_port[1];
        omap->reset_gpio_port[2]    = pdata->reset_gpio_port[2];
        omap->port_mode[0]          = pdata->port_mode[0];
        omap->port_mode[1]          = pdata->port_mode[1];
        omap->port_mode[2]          = pdata->port_mode[2];
        omap->ehci                  = hcd_to_ehci(hcd);
        omap->ehci->sbrn            = 0x20;
        omap->suspended = 0;

        //获得寄存器地址空间
        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        hcd->rsrc_start = res->start;
        hcd->rsrc_len = resource_size(res);
        //进行相应空间的映射
        hcd->regs = ioremap(hcd->rsrc_start, hcd->rsrc_len);

        /* we know this is the memory we want, no need to ioremap again */
        //EHCI 的基本能力寄存器
        omap->ehci->caps = hcd->regs;
        //EHCI 的寄存器基地址
        omap->ehci_base = hcd->regs;
...
        /* get ehci regulator and enable */
        //每个端口的物理电源初始化
        for(i = 0 ; i < OMAP3_HS_USB_PORTS ; i++) {
            if(omap->port_mode[i] != EHCI_HCD_OMAP_MODE_PHY) {
                omap->regulator[i] = NULL;
                continue;
            }
```

```
            snprintf(supply, sizeof(supply), "hsusb%d", i);
            omap->regulator[i] = regulator_get(omap->dev, supply);
            if(IS_ERR(omap->regulator[i])){
                omap->regulator[i] = NULL;
                dev_dbg(&pdev->dev,
                "failed to get ehci port%d regulator\n", i);
            } else {
                regulator_enable(omap->regulator[i]);
            }
        }
        //对 EHCI 使能,主要是时钟等的控制
        ret = omap_start_ehc(omap, hcd);
        if(ret){
            dev_dbg(&pdev->dev, "failed to start ehci\n");
            goto err_start;
        }
        //EHCI 传输相关寄存器
        omap->ehci->regs = hcd->regs
             + HC_LENGTH(readl(&omap->ehci->caps->hc_capbase));

        /* cache this readonly data; minimize chip reads */
        omap->ehci->hcs_params = readl(&omap->ehci->caps->hcs_params);
        //向系统添加总线控制器
        ret = usb_add_hcd(hcd, irq, IRQF_DISABLED | IRQF_SHARED);
        if(ret){
            dev_dbg(&pdev->dev, "failed to add hcd with err %d\n", ret);
            goto err_add_hcd;
        }

        /* root ports should always stay powered */
        ehci_port_power(omap->ehci, 1);
        return 0;
    ...
    }
```

从分析中可见，主要的工作是进行属性的设置、资源的申请和接口的注册，然后通过 usb_add_hcd 的调用实现功能。

相应的 platform device 是在系统初始化时通过 usb_ehci_init 进行的，这里就不进行详述了。

2. 总线传输

总线传输的控制器接口是由 hc_driver 来定义的，DM 3730 ehci 相关的功能由 ehci_omap_hc_driver 来实现，具体细节如下：

```
static const struct hc_driver ehci_omap_hc_driver = {
	.description		= hcd_name,
	.product_desc		= "OMAP - EHCI Host Controller",
	.hcd_priv_size		= sizeof(struct ehci_hcd),
	//总线控制器中断操作
	.irq			= ehci_irq,
	.flags			= HCD_MEMORY | HCD_USB2,

	//总线控制器生命周期的操作
	.reset			= ehci_init,
	.start			= ehci_run,
	.stop			= ehci_stop,
	.shutdown		= ehci_shutdown,

	//管理 IO requests 和相关的资源
	.urb_enqueue		= ehci_urb_enqueue,
	.urb_dequeue		= ehci_urb_dequeue,
	.endpoint_disable	= ehci_endpoint_disable,
	.endpoint_reset		= ehci_endpoint_reset,

	//调度的支持
	.get_frame_number	= ehci_get_frame,

	//root hub 相关操作
	.hub_status_data	= ehci_hub_status_data,
	.hub_control		= ehci_hub_control,
#ifdef CONFIG_PM
	.bus_suspend		= ehci_omap_bus_suspend,
	.bus_resume		= ehci_omap_bus_resume,
#endif
	.relinquish_port	= NULL,
	.port_handed_over	= ehci_omap_port_handed_over,
	.clear_tt_buffer_complete = ehci_clear_tt_buffer_complete,
	.recover_hcd		= ehci_omap_recover_work,
};
```

从中可见，主要的接口都是标准的 EHCI 操作接口。Linux 内核为各种 USB 总线控制器开发了标准的操作接口，EHCI 同样提供了统一的操作接口。EHCI 总线控制器整体架构如图 7-18 所示。图 7-18 引自《EHCI 规范》。

从图 7-18 中可见，EHCI 将传输分为 periodic 和 asynchronous 两个调度队列，中断和周期传输使用 periodic 调度，而其他传输使用 asynchronous 队列。EHCI 的调度和传输都是通过 ehci-q.c 和 ehci-sched.c 来实现的。

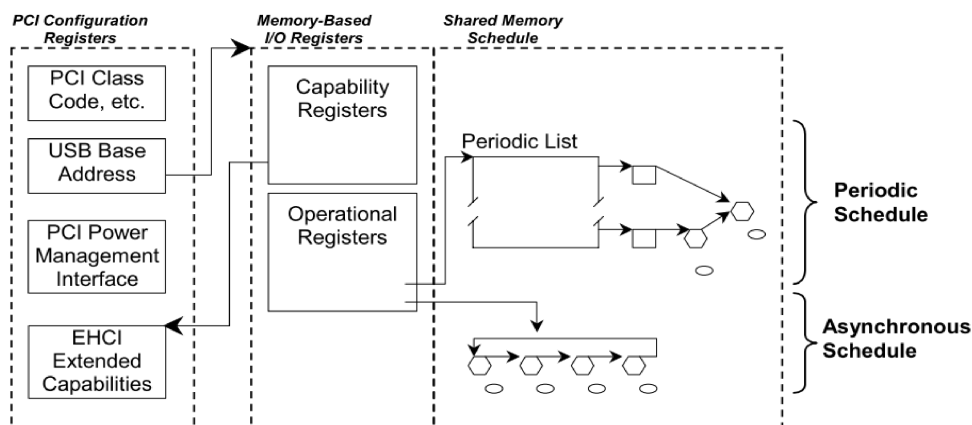


图 7-18　EHCI USB 总线控制器整体框图

EHCI 控制器实现原理：处理器需要把 ISO 数据和 INT 数据建立一张表放在内存中，而 ECHI 的寄存器 FRINDEX 则会跟踪这个表，每一个时间片加 1，FRINDEX 所指之处，控制器就会把这个指针所指向的数据结构中的数据送到总线上去。除了周期调度 EHCI 控制器外，还提拱了另一种调度来处理实时性要求不高的数据，叫做异步调度。处理器把块传输和控制传输的数据按协议要求的数据结构在内存中安排并建立一个链表，AsyncListAddr 则会跟踪这个链表，总线控制器则把 AsyncListAddr 所指向的数据搬运到 USB 总线上去。

EHCI 的实现都是按照 EHCI 控制器规范来做的，其中涉及很多 EHCI 规范的内容，这里就不进行详细分析了。

7.4.4　USB 总线驱动电源管理相关说明

关于 USB 整体的电源管理，首先来看总线部分，在 usb_bus_type 中与电源管理相关的操作接口如下：

```
static const struct dev_pm_ops usb_bus_pm_ops = {
    .runtime_suspend =    usb_runtime_suspend,
    .runtime_resume =     usb_runtime_resume,
    .runtime_idle =       usb_runtime_idle,
};
```

以 usb_suspend_both 为例，其中会通过 usb_suspend_both 对 usb_interface 和 usb_device 进行 suspend 操作，这样就会执行驱动对应的 suspend 操作。usb_device 相应操作由 generic_suspend 来实现，细节如下：

```
static int generic_suspend(struct usb_device *udev, pm_message_t msg)
{
    int rc;
    //若是 root hub 则总线 suspend
    if(!udev->parent)
```

```
            rc = hcd_bus_suspend(udev, msg);
        else if(msg.event == PM_EVENT_FREEZE || msg.event == PM_EVENT_PRETHAW)
            rc = 0;
        else
            rc = usb_port_suspend(udev, msg);

        return rc;
}
```

可见相应设备根据是否为 root hub 进行适合的操作。

以上是 runtime pm 的操作，而对于 SLM 相关操作，则是由 usb_device_type 来实现的，相应的接口是 usb_device_pm_ops：

```
static const struct dev_pm_ops usb_device_pm_ops = {
        .prepare =      usb_dev_prepare,
        .complete =     usb_dev_complete,
        .suspend =      usb_dev_suspend,
        .resume =       usb_dev_resume,
        .freeze =       usb_dev_freeze,
        .thaw =         usb_dev_thaw,
        .poweroff =     usb_dev_poweroff,
        .restore =      usb_dev_restore,
};
```

对于总线控制器的电源管理操作，以 DM 3730 ECHI 为例，主要操作接口如下：

```
static const struct dev_pm_ops ehci_omap_dev_pm_ops = {
        .suspend        = ehci_omap_dev_suspend,
        .resume_noirq       = ehci_omap_dev_resume,
};
```

这里主要是对控制器进行 enable 和 disable 操作。

另外 USB 总线的电源管理操作会调用 hc_driver 的电源管理操作接口，在 DM 3730 EHCI 控制器中是由 ehci_omap_bus_suspend 和 ehci_omap_bus_resume 来实现的，以 suspend 操作为例：

```
static int ehci_omap_bus_suspend(struct usb_hcd *hcd)
{
        struct usb_bus *bus = hcd_to_bus(hcd);
        int ret;
        //停止总线操作,进行相关 remote wakeup 设置
        ret = ehci_bus_suspend(hcd);
        //总线控制器 suspend 操作
```

```
        ehci_omap_dev_suspend(bus -> controller);

        return ret;
    }
```

这样 USB 总线的电源管理部分就基本就完整了。

7.5 小结

本章主要介绍各种总线型设备驱动，各种驱动会根据总线的特点进行设计，并尽量为功能层及应用层提供良好的界面，方便驱动以及应用程序的开发。

总线型驱动各具特色，其中框架设计也包含了很多设计思想，是值得用心学习的部分。这里以层次的方式，将整体的框架从针对应用的上层到总线设备操作的下层进行完整分析。自上而下打通时，整个框架就被比较清晰地展现了。

第 8 章　设备驱动之 SoC 特殊驱动

8.1　SoC 电源管理核心技术详解

8.1.1　SoC 电源管理需求

SoC 处理器中包含各种各样的设备功能模块，设计良好的电源管理设备需要能对每个模块单独进行管理，允许对这些功能模块进行单独的开关操作。并且需要与系统的电源管理功能结合，完成整体的电源管理功能。

总之，SoC 电源管理既要满足系统的整体需求，又要满足功能模块的单独需求，做到兼顾整体和个体。

8.1.2　TI 芯片 SoC 电源管理相关实现详解

SoC 电源管理框架与 SoC 的设计息息相关。Linux 内核在设备模型中提供了 platform 总线来关联 SoC 中的各个子模块，这样就可以利用 platform 总线框架实现 SoC 电源管理的功能。platform 总线相关的电源管理功能已经在设备模型部分进行了介绍。接下来主要以 DM 3730 为例对 TI 芯片 SoC 电源管理实现进行介绍。

DM 3730 SoC 电源管理整体框架如图 8-1 所示。

从图 8-1 可见，DM 3730 SoC 电源管理的核心是 omap_device，其中包含了 platform_device 和 hwmod，platform_device 部分包含了与 platform bus 相关的部分，而 hwmod 包含了 DM 3730 中各个控制器的硬件信息（如时钟、中断、DMA 和片内总线的关联）。这样就将系统框架的 platform 设备到具体芯片内部控制器的信息关联起来，再与具体的操作结合就可以实现电源管理功能。

关于初始化阶段如何将 DM3730 电源管理功能与 platform bus 以及设备模型结合可以参见图 8-2。

从图 8-2 中可见，在各个模块的 init 操作中通过 omap_device_build 来读取 hwmod 的信息并创建相关实体和 platform device，从而与设备模型结合，这样在整个框架上，就完整了。

platform device 的电源管理操作会涉及 platform bus。下面来看看 DM3730 做了什么处理：

```
//DPM，runtime pm 的接口注册初始化程序
static int __init omap_pm_runtime_init(void)
{
    const struct dev_pm_ops *pm;
    struct dev_pm_ops *omap_pm;
```

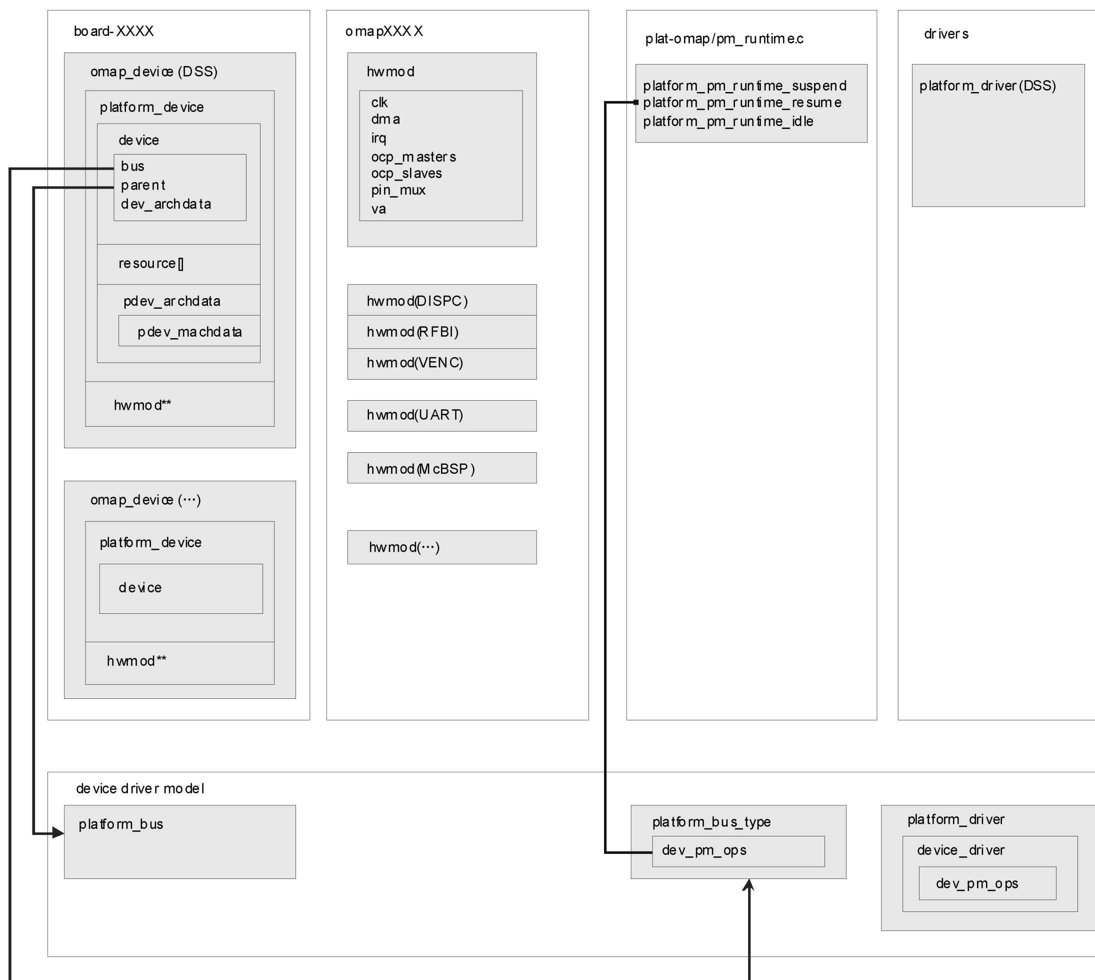


图 8-1　DM3730 SoC 电源管理整体架构

```
//获得 platform bus 的 pm 操作接口
pm = platform_bus_get_pm_ops();
if(!pm){
    pr_err("%s: unable to get dev_pm_ops from platform_bus\n",
        __func__);
    return -ENODEV;
}

//复制所有 platform_bus 的 pm 接口
omap_pm = kmemdup(pm, sizeof(struct dev_pm_ops), GFP_KERNEL);
if(!omap_pm){
    pr_err("%s: unable to alloc memory for new dev_pm_ops\n",
        __func__);
```

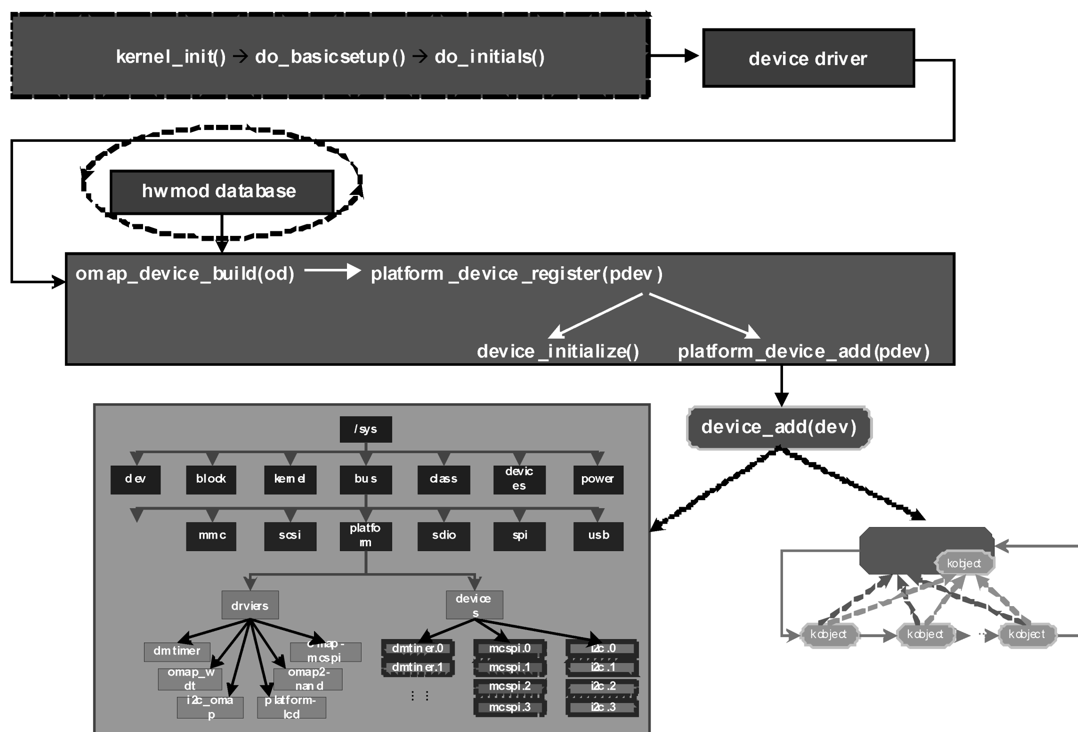


图 8-2 DM3730 SoC 电源管理初始化阶段架构

```
            return - ENOMEM;
        }

        //修改需要修改的接口
        omap_pm -> runtime_suspend = omap_pm_runtime_suspend;
        omap_pm -> runtime_resume = omap_pm_runtime_resume;
        omap_pm -> suspend_noirq = omap_pm_suspend_noirq;
        omap_pm -> resume_noirq = omap_pm_resume_noirq;

        //注册回调保证 porting 为合适的 omap_pm 接口
        platform_bus_set_pm_ops( omap_pm ) ;

        return 0;
    }
```

可见其修改了 platform bus 电源管理操作的部分接口，以 omap_pm_runtime_suspend 和 omap_pm_suspend_noirq 为例看看具体的操作：

```
    //该函数为 runtime power 管理的具体 device 的 suspend 接口。在 omap 中 device
    //为 omap_device
    static int omap_pm_runtime_suspend( struct device  * dev)
```

```
{
    struct platform_device  * pdev = to_platform_device( dev) ;
    int r, ret = 0;

    dev_dbg( dev, "%s\n", __func__) ;

    //首先执行标准的 runtime suspend 操作,需要 device 相应的 driver
    //提供 runtime 的接口
    ret = pm_generic_runtime_suspend( dev) ;

    if( ! ret && dev -> parent == &omap_device_parent) {
        //omap_device 在 build 都会赋值 parent 为 omap_device_parent
        //这里调用 plat_omap 的 omap_device 的 idle 接口,最终
        //调用 omap_hwmod 的 idle 接口,disable clock,并使 module 进入 idle
        r = omap_device_idle( pdev) ;
        WARN_ON( r) ;
    }

    return ret;
};

//在系统进入 suspend 时会调用 suspend_devices_and_enter,先是调用 dpm_suspend,
//使得 driver 的 suspend 被调用,为驱动级别的 suspend;接着调用 dpm_suspend_noirq,
//使得 bus 的 pm 中的 suspend_noirq 被调用,为 hwmod 级别的 suspend
int omap_pm_suspend_noirq( struct device  * dev)
{
    struct device_driver  * drv = dev -> driver;
    int ret = 0;

    if( ! drv)
        return 0;

    //如果有调用驱动的相关函数
    if( drv -> pm) {
        if( drv -> pm -> suspend_noirq)
            ret = drv -> pm -> suspend_noirq( dev) ;
    }

    / *
     * The DPM core has done a' get' to prevent runtime PM
     * transitions during system PM.  This put is to balance
     * out that get so that this device can now be runtime
```

```
    *  suspended.
    */
  //这里可调用 runtime 的接口完成后续操作,同步执行 driver suspend 的
  pm_runtime_put_sync(dev);

  return ret;
}
```

可见主要还是围绕 omap device 的操作来执行。而 omap device 会调用 omap_device_pm_latency 中的操作，细节如下：

```
struct omap_device_pm_latency {
    u32 deactivate_lat;
    u32 deactivate_lat_worst;
    int( * deactivate_func)(struct omap_device  * od);
    u32 activate_lat;
    u32 activate_lat_worst;
    int( * activate_func)(struct omap_device  * od);
    u32 flags;
};
```

这里既包括了操作也包括了 pm QoS 的内容，实现了比较全的功能。相应的操作接口如下：

```
int omap_device_idle_hwmods(struct omap_device  * od);
int omap_device_enable_hwmods(struct omap_device  * od);
```

整体的电源管理初始化由 omap3_pm_init 完成，详细内容如下：

```
//power management 相关的初始化处理函数非常重要
static int __init omap3_pm_init(void)
{
    struct power_state  * pwrst,  * tmp;
    struct clockdomain  * neon_clkdm,  * per_clkdm,  * mpu_clkdm,  * core_clkdm;
    int ret;

    if( ! cpu_is_omap34xx())
        return -ENODEV;

    //首先检查相应的 errata
    pm_errata_configure();

    printk(KERN_ERR "Power Management for TI OMAP3. \n");
```

```
	//virtual terminal 可以在 suspend 时创建新的 console
	//要 disable 该功能,通过 vt 的接口函数 disable 该功能
#ifdef CONFIG_OMAP3_PM_DISABLE_VT_SWITCH
		pm_set_vt_switch(0);
#endif

	/* XXX prcm_setup_regs needs to be before enabling hw
	 * supervised mode for powerdomains */
	//在 enable 每个 power domain 的 hwsup 功能之前要初始化 prcm 的寄存器
	prcm_setup_regs();

	//做 prcm 中断处理函数的初始化
	ret = request_irq(INT_34XX_PRCM_MPU_IRQ,
			(irq_handler_t)prcm_interrupt_handler,
			IRQF_DISABLED, "prcm", NULL);
	if(ret){
		goto err1;
	}

	//对于每个 power domain 执行 setup 初始化
	ret = pwrdm_for_each(pwrdms_setup, NULL);
	if(ret){
		printk(KERN_ERR "Failed to setup powerdomains\n");
		goto err2;
	}

	//对于每个 clock domain 执行 setup 初始化
	(void) clkdm_for_each(clkdms_setup, NULL);

	mpu_pwrdm = pwrdm_lookup("mpu_pwrdm");
	if(mpu_pwrdm == NULL){
		printk(KERN_ERR "Failed to get mpu_pwrdm\n");
		goto err2;
	}

	//记录必要的 power domain 和 clock domain,以便后续操作
	neon_pwrdm = pwrdm_lookup("neon_pwrdm");
	per_pwrdm = pwrdm_lookup("per_pwrdm");
	core_pwrdm = pwrdm_lookup("core_pwrdm");
	cam_pwrdm = pwrdm_lookup("cam_pwrdm");

	neon_clkdm = clkdm_lookup("neon_clkdm");
```

```
mpu_clkdm = clkdm_lookup("mpu_clkdm");
per_clkdm = clkdm_lookup("per_clkdm");
core_clkdm = clkdm_lookup("core_clkdm");

//对 sleep 代码进行 internal RAM 的加载
omap_push_sram_idle();

//注册 ldm suspend 的操作接口
suspend_set_ops(omap_pm_ops);

//临时 system idle 的 hook 注册
//后续 omap3_idle_init 中对 cpuidle_device 注册时会再次通过
//cpuidle_install_idle_handler 注册该 system idle 的 hook 即 pm_idle
pm_idle = omap3_pm_idle;
omap3_idle_init();

/*
 * RTA is disabled during initialization as per erratum i608
 * it is safer to disable RTA by the bootloader, but we would like
 * to be doubly sure here and prevent any mishaps
 */
//根据勘误,omap3630 要 disable memory 的 retention till access 的功能
//避免 deepsleep 有问题
if(IS_PM34XX_ERRATUM(PM_RTA_ERRATUM_i608))
    omap3630_ctrl_disable_rta();

//将 neon 加 MPU 的依赖,这样 neon 可以依赖于 MPU 进入不同的状态
clkdm_add_wkdep(neon_clkdm, mpu_clkdm);

//如果 security chip,初始化的时候保护相应的 secure ram 的上下文
if(omap_type() != OMAP2_DEVICE_TYPE_GP){
    omap3_secure_ram_storage =
        kmalloc(0x803F, GFP_KERNEL);
    if(!omap3_secure_ram_storage)
        printk(KERN_ERR "Memory allocation failed when"
                "allocating for secure sram context\n");

    local_irq_disable();
    local_fiq_disable();

    omap_dma_global_context_save();
    omap3_save_secure_ram_context();
```

```
            omap_dma_global_context_restore();

            local_irq_enable();
            local_fiq_enable();
        }

        //对 pm debug 信息,关于寄存器的信息进行初始化
        pm_dbg_regset_init(1);
        pm_dbg_regset_init(2);

        //重要 scratchpad memory 的内容保存,以便 deep sleep 后从系统恢复时用
        omap3_save_scratchpad_contents();
    err1:
        return ret;
    err2:
        //如果错误则释放 prcm 中断及 power state
        free_irq(INT_34XX_PRCM_MPU_IRQ, NULL);
        list_for_each_entry_safe(pwrst, tmp, &pwrst_list, node){
            list_del(&pwrst->node);
            kfree(pwrst);
        }
        return ret;
    }
```

这里除了信息加载外，最重要的就是调用 omap_push_sram_idle 与操作 suspend_set_ops(omap_pm_ops)。omap_push_sram_idle 是将系统进入待机状态的最后操作以及从待机状态恢复的开始操作的代码放入片内 RAM 中，这样才能保证系统正确进入低功耗状态并且能正确恢复；而 suspend_set_ops(omap_pm_ops)是修改系统设备模型中 SLM 整体进入待机状态的操作接口，详细内容如下：

```
    //该接口为 omap3 针对 ldm suspend 提供的 ops 接口,使系统可以进入相应的 suspend
    //状态,目前支持 suspend 的只有 STANDBY 和 MEM 状态
    static const struct platform_suspend_ops omap_pm_ops[] = {
        {
            .begin          = omap3_pm_begin,
            .end            = omap3_pm_end,
            .enter          = omap3_pm_enter,
            .valid          = suspend_valid_only_mem,
        }
    };
```

在 omap3_pm_enter 中会通过 omap3_pm_suspend 来完成最后的操作，细节如下：

```
//该函数为 ldm suspend 的 hook 接口之一提供进入 suspend 实际操作接口
static int omap3_pm_suspend(void)
{
    struct power_state *pwrst;
    int state, ret = 0;

    //如果需要 timer 唤醒则设置
    if(wakeup_timer_seconds || wakeup_timer_milliseconds)
        omap2_pm_wakeup_on_timer(wakeup_timer_seconds,
                                 wakeup_timer_milliseconds);

    /* Read current next_pwrsts */
    //读取每个 power domain 的当前状态以便恢复
    list_for_each_entry(pwrst, &pwrst_list, node)
        pwrst->saved_state = pwrdm_read_next_pwrst(pwrst->pwrdm);
    /* Set ones wanted by suspend */
    list_for_each_entry(pwrst, &pwrst_list, node){
        //设置每个 power domain 要进入的 state
        if(omap_set_pwrdm_state(pwrst->pwrdm, pwrst->next_state))
            goto restore;
        //clear powerdomain 的 pre power state 以准备 sleep
        if(pwrdm_clear_all_prev_pwrst(pwrst->pwrdm))
            goto restore;
    }

    //uart prepare suspend
    omap_uart_prepare_suspend();
    //清除 pending 的 irq,使系统进入 sleep 模式
    omap3_intc_suspend();

    //系统真正进入 sleep 的 idle 操作实现
    omap_sram_idle();

restore:
    /* Restore next_pwrsts */
    //到这里系统已经恢复,需要做必要的检查
    list_for_each_entry(pwrst, &pwrst_list, node){
        //读取 pre power state,以清楚 power domain 从哪个 state 被唤醒
        state = pwrdm_read_prev_pwrst(pwrst->pwrdm);
        if(state > pwrst->next_state){
            //唤醒后的 pre power state 比想要进入的 power state 大,
            //说明没有能进入希望的 state,输出 log,以引起注意
```

```
                printk(KERN_INFO "Powerdomain(%s) didn't enter "
                        "target state %d\n",
                        pwrst->pwrdm->name, pwrst->next_state);
                ret = -1;
            }
            //恢复 power domain 进入 sleep 之前的 state
            omap_set_pwrdm_state(pwrst->pwrdm, pwrst->saved_state);
        }
        if(ret)
            printk(KERN_ERR "Could not enter target state in pm_suspend\n");
        else
            printk(KERN_INFO "Successfully put all powerdomains "
                    "to target state\n");

        return ret;
    }
```

至此，对 DM3730 芯片电源管理整体实现进行了完整说明。该架构整体比较清晰，将共性的部分做了完整抽象，从而简化了各个功能模块的实现过程。

该框架可以在 DM 3730 中使用，TI 的其他系列芯片（包括 DM81XX 和 Sitara 系列）也都可以使用该架构实现电源管理功能。

8.2 小结

本章针对 SoC 电源管理技术展开，主要介绍了 DM3730 SoC 电源管理的设计与实现。电源管理技术是一个复杂的工程，需要考虑从设备到内部模块，从整体功能到个体功能的各个部分。要设计一个良好的 SoC 电源管理架构，就要从内核的整体需求出发，将片内各个模块的共性抽象形成统一的管理实体，再与具体的芯片操作结合，从而形成良好的设计。在这些方面，TI 芯片（特别是 DM3730）做了很多创造性的工作，值得学习、研究。

参 考 文 献

［1］ Wolfgang Mauerer. 深入 Linux 内核架构[M]. 郭旭，译. 北京：人民邮电出版社，2010.

［2］ Daniel P Bovet，Marco Cesati. 深入理解 Linux 内核［M］. 3 版. 陈莉君，张琼声，张宏伟，译. 北京：中国电力出版社，2007.

［3］ Texas Instruments. AM/DM37x Multimedia Device Technical Reference Manual[EB/OL]，2012.

［4］ Texas Instruments. TMS320DM816x DaVinci Digital Video Processors Technical Reference Manual[EB/OL]，2013.

［5］ Texas Instruments. AM335x ARM Technical Reference Manual［EB/OL］，2013.

［6］ Linux 内核源码分析——内核启动. http://blog. chinaunix. net/uid－20543672－id－3018233. html.

［7］ 陈学松. 深入 Linux 设备驱动程序内核机制[M]. 北京：电子工业出版社，2012.

在线互动交流平台

官方微博：http://weibo.com/cmpjsj
豆 瓣 网：http://site.douban.com/139085/
读者信箱：cmp_itbook@163.com

深入剖析Linux内核与设备驱动

内容提要

本书从需求的角度出发，以层次分析的方法探究Linux内核以及驱动的各部分框架和实现；以TI的嵌入式芯片为例，对内核各部分功能特别是电源管理功能进行代码级别的分析；以需求是什么、如何实现相应需求来进行整体分析，更易于对系统的把握和理解。本书将软件和硬件结合起来详细分析了嵌入式处理以及Linux内核实现中的各种技术。

本书可以作为本科和研究生操作系统的参考书，也适合Linux内核、驱动以及嵌入式系统各种级别的开发者和爱好者阅读。

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
服务咨询热线：010-88379833
读者购书热线：010-88379469
网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
金 书 网：www.golden-book.com
教育服务网：www.cmpedu.com
封面无防伪标均为盗版


机工教育微信服务号

上架指导 计算机/Linux
ISBN 978-7-111-49426-3
策划编辑◎时 静 / 封面设计◎



ISBN 978-7-111-49426-3
9 787111 494263
定价：99.80元